# Workshop: W03

The First International Workshop on Automated Program Analysis, Testing and Verification.

# The First International Workshop on Automated Program Analysis, Testing and Verification

**Introduction** ..... 1

**Testing 1** ..... 2

Predicting the effectiveness of evolutionary testing for the measurement ..... 3
of extreme execution times.
*H. Gross, B. Jones and D. Eyres*

Automated testing of real−time tasks ..... 9
*J. Wegener, R. Pitschinetz and H. Sthamer*

Automated evaluation of COTS components ..... 11
*C. Mueller and B. Korel*

Automating the testing of databases ..... 15
*R. Davies, R. Beynon and B. Jones*

**Static Analysis** ..... 21

ATGen: Automatic test data generation using constraint logic ..... 22
programming and symbolic execution
*C. Meudec*

Program analysis and test hypotheses complement ..... 32
*R. Hierons and M. Harman*

Annotation−assisted lightweight static checking ..... 40
*D. Evans*

Analyzing dependencies in java bytecode ..... 43
*J. Zhao*

**Testing 2** ..... 50

Third eye − Specification−based anlysis of software execution traces ..... 51
*R. Lencevicius, A. Ran and R. Yairi*

Testing, proof and automation: An integrated approach ..... 57
*S. Burton, J. Clark and J. McDermid*

Test Generation and Recognition with Formal Methods ..... 64
*P. Ammann and P. Black*

A formally founded componentware testing methodology ..... 68
*K. Bergner, H. Lötzbeyer and A. Rausch*

**Dynamic Analysis** ..... 73

Java model checking ..... 74
*D. Park, U. Stern, J. Skakkebæk and D. Dill*

On the specification and semantics of source level properties in java ..... *83*
R. Iosif and R. Sisto

Towards synergy of finite state verification and testing ..... 89
*G. Naumovich and P. Frankl*

**Testing 3** ..... 95

Test automation for object−oriented frameworks ..... 96
*M. Schnizler and H. Lichter*

Object−oriented specification−based testing using UML statechart diagrams .. 101
*M. Vieira, M. Dias and D. Richardson*

A framework for practical, automated black−box testing of ..... 106
component−based software
*S. Edwards*

Towards the determination of sufficient mutant operators for C ..... 115
*E. Barbosa, J. Maldonado and A. Vincenzi*

# The First International Workshop on Automated Program Analysis, Testing and Verification

Welcome to the first international workshop on automated program analysis, testing and verification. This workshop aims to bring together researchers and developers interested in automated verification, analysis and testing of software to build new collaborations and increase the level of co−operation between the communities.

**What the workshop has to offer.**
The workshop has attracted more than 25 submissions of which 19 have been selected to appear in these proceedings and 16 have been selected for presentation over the two days. The workshop will also feature two keynote speakers − G. Holzmann and M. J. Harrold. The workshop will be structured as a number of mini−panel sessions, consisting of 3 fifteen minute presentations followed by forty−five minutes of questions and discussion. The aim of this format is to prompt detailed discussions.

**Acknowledgements.**
We would like to thank all those who have contributed submissions to this workshop. We would also like to thank the program committee members who have put in many hours work reviewing the submissions.

**Program Committee Members.**

| | |
|---|---|
| Jay Corbett | University of Hawaii |
| Dennis Dams | University of Eindhoven |
| David Dill | Stanford University |
| Matt Dwyer | Kansas State University |
| Patrice Godefroid | Lucent Technologies |
| John Hatcliff | Kansas State University |
| Klaus Havelund | NASA Ames Research Center |
| Gerard Holzmann | Lucent Technologies |
| Bryan Jones | University of Glamorgan |
| Bogdan Korel | Illinois Institute of Technology |
| Jim Larus | Microsoft Research |
| Rustan Leino | Compaq Systems Research Center |
| Tom Reps | University of Wisconsin |
| Debra Richardson | University of Califorina, Irvine |
| Riccardo Sisto | Polytechnic of Torino |
| Daniel Weise | Microsoft Research |
| Martin Woodward | University of Liverpool |

We hope you enjoy the workshop!

John Penix, Nigel Tracey and Willem Visser
Organising Committee

# Testing 1

**Predicting the effectiveness of evolutionary testing for the measurement of extreme execution times.**

**H-G Gross, B F Jones, D E Eyres.**
**School of Computing, University of Glamorgan, Pontypridd, CF37 1DL, UK.**
**Bfjones@glam.ac.uk**

**Abstract.**
Evolutionary algorithms have been used to generate tests automatically to measure the worst and best case execution times for software. Metrics are proposed to predict the effectiveness of evolutionary algorithms; the metrics are based on complexity measured in terms of decisions weighted by nesting level. Other factors are the cohesion and coupling of the module and the filtering effect of predicates that have only a small probability of execution.

**Introduction.**

The worst and best case execution times (W/BCET) of a real-time system are important since the system must produce results according to a specified time schedule. W/BCET analysis demands full knowledge about the behaviour of the underlying hardware, the scheduling and timing of the operating system, and the execution time of the real-time software under development. Software testing is a widely used and accepted technique for verification and validation and is considered the ultimate check on the conformance of the software to its specification.

Evolutionary testing (ET) is a new testing technique based upon the application of evolutionary algorithms (EA) to the generation of test sets. This technique has already been successfully applied to structural testing and for testing the timing behaviour of systems. The latter complements static analysis for timing.

Evolutionary testing is based on a typical search/optimisation technique (EA) and such techniques seldom reveal any information on how close the best solution of the search process comes to the actual optimal solution. This lack of quality assessment of the testing process might have inhibited its widespread use. Experiments with evolutionary testing performed for this work revealed a correlation between the success of the search technique to find the optimal (or near optimal) solution and the complexity of the test object. Complexity is difficult to define in the context of software. Many different interpretations can be found in the literature ranging from "difficulty to maintain, change and understand software" to "amount of information which must be understood and processed in order to produce, use maintain and change software".

The key factors in ensuring that ET executes efficiently and effectively are the choice of how to represent the input test set, how to calculate the fitness of the solution associated with the test set, and to decide when to stop the search. There is no guarantee that ET will find the W/BCET in a reasonable time or even if it will ever find it. The aim of this work is to devise a metric to predict whether ET is an appropriate technique for the software under investigation.

**Complexity measures for evolutionary testing**

Many definitions of software complexity emphasise cognitive complexity which indicates the effort needed to understand the software based on control flow or data flow. Most software complexity metrics, including the standard metrics of Halstead, McCabe, Myers or Harrison concentrate on complexity as understandability. Software testability which can be seen as the degree of difficulty to test software, is related to software complexity, and most definitions of software testability focus on one of its properties. For example Bache and Mullerburg use the terminology to calculate the number of required test cases for a test object for satisfying a specific test strategy.

They define this as the effort to test software, although it is not the same as 'software testability'.

A very specific definition of an objective complexity measure, and consequently testability, is required for evolutionary testing. A possible definition of complexity could be "the difficulty for the testing process (EA) to generate test cases which satisfy the test criterion". On a module level, complexity may be seen as the sum of all program properties which make it difficult for an evolutionary algorithm to generate input parameters corresponding to the test objective, for instance finding the worst-case execution time. In this case, testability can be defined as the degree of difficulty to successfully apply evolutionary testing to a particular module. Here, complexity must be understood, not from the human perspective, but primarily from the perspective of the (automatic) testing methodology which is, in this case, an optimisation algorithm.

The execution of a program under test ideally covers every single entry-exit path which results in full path-coverage. This can be regarded as a reasonable strategy for testing the timing behaviour of real-time software since all program sections must be executed in order to determine their execution time. Consequently, the testing process must be able to examine all possible paths in order to 'decide' which of them are most promising for the required testing objective. The difficulty of generating input according to this requirement is determined by the decisions in the test program, and here, 'difficult' decisions create serious problems for evolutionary testing. For instance, these can be decisions that create small domains so that the branch is only taken with a very low probability.

The following list outlines properties of test programs which have been identified as creating most problems for an evolutionary testing process to generate test cases for finding the B/WCET:

- *High parameter interdependence and/or large input vectors*. High interdependence may either be caused by decisions which require some of the input to be in a specific relation, for example a specified pattern, in order to lead the program flow into a distinct branch or by calculations on input. In the second case the values of the input variables determine the time it takes to perform the calculations.

- *Small input domains or single-value domains*. These are caused by decisions that execute one branch with a very low probability. This restricts the ability of the EA for large search spaces as it is unlikely to generate the required value by chance.

- *Parameter dependent loops*. Loops whose number of iterations depend upon input variables are equivalent to decisions with a single-value domain. Here, the EA must generate input that leads to the lowest or highest number of iterations for the loop.

- *Nesting and sequencing*. Combinations of all previous items.

**A simple structural complexity measure**

Determining the nesting/sequencing of a program can be regarded as an initial step towards a structural predictive measure of ET performance. A number of nesting measures exist but those based upon a program's flowgraph are particularly useful for ET; each node is assigned a weighting that can be used to specify the node complexity more accurately. The complexity may be defined as:

$$BAND = \sum_{i=1}^{N} L(i) * n(i)$$

where $N$ is the number of nodes in the flowgraph, $n(i)$ is the weight of node $i$

(typically *n(i)* = 1) and *L(i)* is the nesting level of node *i*. This captures the total nesting/sequencing of a program. From the measure *BAND* we derive the measure *Essential BAND* (*ESS-BAND*) which only considers the decision nodes in a flowgraph. This measure defines the essential nesting (or decision nesting/sequencing) of a program. The measure is defined as

$$ESS - BAND = \sum_{i=1}^{D} L(i) * n(i)$$

where *D* is the number of decision nodes (branches/loops) in the flowgraph. This can be regarded as a simple basic structural measure which is sufficient to indicate the ability of an EA to find input parameters for a test object corresponding to full node coverage (every node is visited at least once). However, it is not sufficient for a test object containing parameter dependent loops or single-value domains. For these test objects, the weight *n(i)* in measures *BAND* and *ESS-BAND* is set to *n(i)* = 2. The performance of the search technique dropped by about 20-50 % depending on the nesting level on which the conditional was inserted and depending on the size of the search space. We believe that a factor of 2 for each of these critical nodes captures this effect accurately enough.

**Measurements and testing performance results.**

The genetic algorithm used in the experiments for this work was developed by the authors and is intentionally simple and fixed. Although there exist specific evolutionary operators for some test object classes which improve the overall outcome, this prohibits a comparison of the reaction of the search technique on the different test objects. The aim is to determine whether a particular class of software is appropriate for timing analysis using ET rather than to optimise the genetic operators. The following genetic operators were used throughout the experiments:

- Population size = 40 (keep forty best individuals).
- Tournament selection, tournament size = 4.
- Discrete recombination (uniform crossover $p_c = 0.5$).
- Low constant mutation rate ($p_m = 0.001$).
- Rank Based Fitness.
- Random initialisation of the chromosomes.

These values were kept constant during the experiments. The testing terminated when the fitness had not improved for 200 generations. For each module under investigation, the path corresponding to the WCET was determined by inspection. The fitness function for each test set generated by the EA was the achieved percentage coverage of the worst case execution path. This was measured by instrumentation of the source/object code. The condition for using a test module in these experiments is that its actual worst-case execution path can be analysed/retrieved. Even for many moderately complex modules this is an extremely difficult task and provides the motivation for this work. The test objects are three simple sorting algorithms, a few modules taken from a graphical contour plotting package and some taken from a robot vision system (see table 1).

Table 2 displays the properties of the test objects and their measures *Essential BAND* (*ESS-BAND*) and the success rate of the evolutionary search (last column in table 2). The control flow path corresponding to the WCET is known for the modules under investigation and the last column is the percentage of nodes covered during the evolutionary testing process. It is not to be confused with the measured WCET of the

test module compared to the actual maximum execution time. This relation depends upon the execution time complexities between the decision nodes for which no measure is defined yet.

The modules in table 2 are ordered according to their ET-success rate measured as the percentage of the worst case execution time path covered. They are compared to the measured *ESS-BAND*. In general, *ESS-BAND* is capable of indicating the success of the evolutionary testing process quite accurately for many cases. The overall correlation of measure and ET performance of the test objects is given in figure 1 which can be used to predict the evolutionary testing success rate for a new module. However, a few cases occurred during the experiments where the correlation was poor. These are exceptions which must be further investigated. The module *polex1* is a redesigned version of the original module *polex*. The original never executes the longest path despite its low structural complexity. It violates an important principle of 'good software design': low coupling. The input vector of the original module is 1080 bytes long with only 18 bytes actually accessed. This creates an insurmountable difficulty for the evolutionary search as the probability of the EA changing one of the 18 bytes is slim. Changing the design and reducing the size of the input vector leads to the generation of the worst-case execution path. This corresponds to the low complexity of this new module *ESS-BAND*= 2(table 2).

The poor performance of the search technique for module *dzz* is caused by a violation of the principle of high cohesion. By investigating the structure of this module, three unrelated aspects of functionality were identified. A new design of this original module resulted in three new modules: *dzz1, dzz2* and *dzz3*. The new units could be implemented much more simply (*ESS-BAND*= 6 and 10 compared to *ESS-BAND*= 65 for the original unit) so that the performance of the search technique increased dramatically for modules *dzz1*= 100% and *dzz3*= 100%. However, for the second module this drastic improvement was not observed. This is due to extremely narrow domains for two decision nodes in the flowgraph. These two nested decisions only evaluate to true for 30 out of 65536 values in order to follow the longest execution path. This narrow domain is not captured by the measured *ESS-BAND*. It is easy to see for humans why the outcome of this module is so poor but at this stage it is not quite clear how to implement an automatic tool which would capture this type of complexity.

ET performs as expected for the three sorting algorithms *is, bs1* and *bs2*. Their complexity *ESS-BAND* is only 6 but repeated iterations have a filtering effect and there is a high parameter interdependence. Each value of a list for a sorting algorithm must be in a defined relation to each other value in the list. By increasing the size of the input vector, the evolutionary search becomes increasingly more difficult and the overall performance decreases.

The size of the input domain and the complexity of a test object are inversely proportional. For increasing structural complexity and decreasing size of the input vector the outcome of evolutionary testing stays constant for many cases. Therefore, simple modules with huge input vectors can be tested just as easily and successfully as large and complex test objects with moderately sized input vectors. For the design of new modules it means that keeping the input simple allows a larger and more complex structure. *ESS-BAND* does not currently measure this effect, but it would be very useful to find a way to assess this.
A similar topic is assessing the filtering effect which is imposed on the system through high input interdependence. This effect could probably be measured by looking at operations which are carried out on the input vector between and in the decision nodes. Although, it is not quite

clear at this stage how to define rules for this assessment as it is a mentally demanding process which is not easy to implement as an algorithm for an automatic testing environment.

**Conclusion.**

There is a strong correlation between the complexity of software as measured by *ESS-BAND* and the efficiency of ET. There is evidence that the performance of ET deteriorates

- when the software module has been designed with low cohesion and high coupling;
- when node predicates depend on input variables that are also changed in the software, and
- when branches have only a small probability of execution.

| Module name | Module Description |
|---|---|
| bs1 | Bubble Sort for list of four-byte integers. |
| bs2 | Bubble Sort for list of four-byte integers. |
| is | Insertion Sort for list of four-byte integers. |
| polex | Contour Plotting - noise filter. |
| polex1 | Contour Plotting - redesigned polex. |
| delsing | Contour Plotting - noise filter. |
| epd | Contour Plotting - extrapolation. |
| dzz | Contour Plotting - noise filter. |
| dzz1 | Contour Plotting - redesigned dzz part 1. |
| dzz2 | Contour Plotting - redesigned dzz part 2. |
| dzz3 | Contour Plotting - redesigned dzz part 3. |
| di | Robot Vision - difference of two picture frames. |
| sobel | Robot Vision - edge detector. |
| min | Robot Vision - filter. |
| median | Robot Vision - filter. |

Table 1: Description of the test objects.

| Module | N | D | BAND | ESS-BAND | Search Space (bytes) | ETsuccess (%) |
|---|---|---|---|---|---|---|
| Polex1 | 13 | 1 | 15 | 2 | 18 | 100.0 |
| Dzz1 | 6 | 4 | 14 | 6 | 1080 | 100.0 |
| Dzz3 | 5 | 3 | 11 | 6 | 1080 | 100.0 |
| sobel | 9 | 2 | 16 | 3 | 1024 | 96.9 |
| di | 6 | 2 | 11 | 3 | 1024 | 95.9 |
| Bs2 | 10 | 3 | 26 | 6 | 1024 | 95.9 |
| is | 9 | 3 | 21 | 6 | 1024 | 95.8 |
| Bs1 | 7 | 3 | 19 | 6 | 1024 | 95.6 |
| min | 9 | 4 | 21 | 10 | 1024 | 87.2 |
| polex | 17 | 3 | 29 | 6 | 1080 | 80.0 |
| median | 15 | 4 | 39 | 10 | 1024 | 80.0 |
| Dzz2 | 9 | 4 | 25 | 10 | 1080 | 5.0 |
| delsing | 6 | 4 | 23 | 19 | 1080 | 65.6 |
| epd | 9 | 6 | 42 | 27 | 1080 | 43.5 |
| dzz | 20 | 12 | 99 | 65 | 1080 | 1.0 |

Table 2: Properties of the test objects. N corresponds to the nodes in the flowgraph, D is the number of decision nodes. ET-success is the performance of the search technique. It can be interpreted as the ability to generate test data to cover the path which leads to the longest execution time. A value of 100 % for ET-success means that we cover all the nodes on the control flow path for the longest execution path for this module.

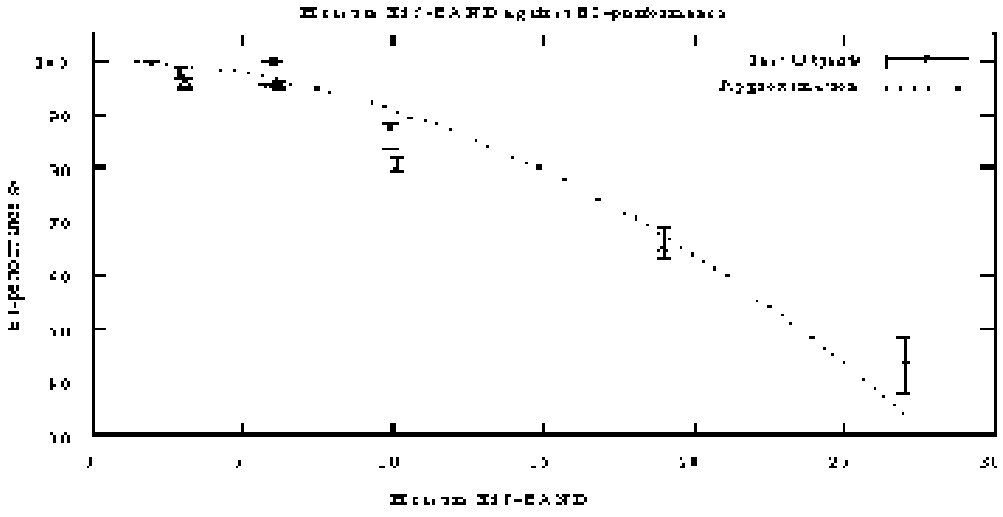


Figure 1: *ESS-BAND* complexity versus average ET performance for ten tests for the modules *polex1, dzz1,dzz3, sobel, diff, bs2, is, bs1, min, median, delsing* and *epd.*

First International Workshop on Automated Program Analysis, Testing and Verification

# Automated Testing of Real-Time Tasks

Joachim Wegener, Roman Pitschinetz, Harmen Sthamer
DaimlerChrysler AG, Research and Technology, Alt-Moabit 96a, D-10559 Berlin, Germany
Joachim.Wegener@daimlerchrysler.com
Roman.Pitschinetz@daimlerchrysler.com
Harmen.Sthamer@daimlerchrysler.com

The development of embedded systems is a crucial area of responsibility in industrial practice. Many embedded systems need to meet real-time requirements. This adds a new dimension to the testing of such systems – not only the logical behavior, but also the temporal behavior of these systems requires thorough testing. In comparison with conventional software systems, the testing of embedded systems is more complex due to several specific technical characteristics such as the development in host-target environments, the intense interaction of the systems with the real application environment, and the limited resources of the target system. In order to facilitate systematic and largely automated testing in defiance of the complexity of real-time systems powerful testing tools are required. Therefore, in this work the testing system TESSY has been extended in order to support the total testing life-cycle of real-time tasks. New components allow a thorough examination of the logical as well as the temporal behavior of the tasks. The logical behavior is tested by means of function-oriented and structure-oriented testing methods; the testing of temporal behavior is automated by evolutionary testing.

TESSY [1] concentrates mainly on test case design, test execution, monitoring, test evaluation, and test documentation. TESSY automates all test activities except the test case generation for examining logical program behavior. In order to automate the test execution, the required test drivers are generated, communication between host and target system is automatically built, the program code is instrumented and coverage analysis is performed, and the execution times on the target system are measured. Regression testing is also entirely automated by TESSY.

For the generation of functional test cases, TESSY uses the classification-tree method [2]. TESSY therefore, contains the classification-tree editor, CTE [3]. Branch testing is supported by the structure-oriented test method. It is possible to instrument the program code to record the branches executed during functional testing and to define the amount of branch coverage obtained. On the basis of this information, the functional test may be further improved or expanded by structure-oriented test cases. This test strategy guarantees an extensive test of the logical program behavior. The test can be run with or without instrumentation in order to exclude side-effects from the instrumentation. The results generated from the test object will then be automatically compared with each other and deviations documented in the generated test documentation.

The most important property, however, is the automation of testing temporal behavior by means of evolutionary testing. Errors in the temporal behavior of real-time systems usually result from a violation of specified timing constraints. The tester's task is to find input situations that result in the maximum execution times. If the execution times exceed the specified constraints, an error has been detected. In evolutionary testing the search for the longest execution time is considered a discontinuous, nonlinear optimization problem, with the input domain of the test object as search space, sets of test data as decision variables, and execution times as objective values. In order to solve this optimization problem, evolutionary algorithms are used to approximate the longest execution times of a test object within several generations. The application of evolutionary algorithms for test data generation is known as evolutionary testing. Previous works have shown that evolutionary testing is superior to random testing [4] and systematic testing [5] when it comes to examining the temporal behavior of real-time systems.

Logical and temporal behavior testing are combined through *seeding*. Test data collected by the tester for the functional test are integrated into the initial population of the evolutionary test. This means that the evolutionary test benefits from the tester's knowledge concerning the functions and internal structures of the test object. The search does not commence with a randomly generated population.

The first industrial application of TESSY with the set of properties described in this paper was initiated last year for testing an engine control system containing more than 20 different tasks. All tasks were tested for their logical program behavior with the classification-tree method and complete branch coverage for all the tasks was reached. Further, six time-critical tasks have been tested for their temporal behavior with evolutionary testing. To avoid probe effects (deviations from actual run-time behavior) instrumentation is turned off for the tasks.

The number of input parameters of these tasks varies from 9 to 18 with a number of program lines set between 39 and 119, the static program paths differ from 1 to 37 million and the cyclomatic complexity from 1 to 27. For each task evolutionary testing generated between 7,500 and 15,000 sets of test data. The target processor is the Siemens C167 with 1 Mbytes SRAM and with a speed of 20 MHz. The testing of one single task took approximately 1 hour and all tests were carried out on the target system that has been designated for future use in cars. The execution times were determined using hardware timers of the target environment with a resolution of 400 ns. The results of the evolutionary tests compared with the execution times determined by the developers' tests are shown in Table 1.

| task | Longest execution time in µs | | Lines of code | No. of parameters | Program paths | Cyclomatic Complexity |
|---|---|---|---|---|---|---|
| | Evolutionary test | Developer test | | | | |
| 1 | 69,6 µs | 67,2 µs | 41 | 18 | 224 | 10 |
| 2 | 120,8 µs | 108,4 µs | 119 | 18 | 37.748.736 | 27 |
| 3 | 112,0 µs | 108,4 µs | 98 | 17 | 1 | 1 |
| 4 | 68,8 µs | 64,0 µs | 81 | 32 | 2 | 2 |
| 5 | 59,6 µs | 57,6 µs | 39 | 14 | 408 | 11 |
| 6 | 58,4 µs | 54,0 µs | 56 | 9 | 63.864 | 18 |

Table 1: Maximum execution times of engine control tasks determined by evolutionary testing and developers' tests

These TESSY extensions described have proved to be highly applicable in practice for testing an engine control system. Both, the logical and the temporal behavior have been thoroughly tested. The deployment of the CTE methodology has been approved and utilized by the developers in order to generate systematic test cases that obtained 100% branch coverage. All other test activities are fully automatically executed on the target system, specifically the testing of the temporal behavior. For the 6 tasks testing the temporal behavior, longer execution times were found with the evolutionary test than with the developers' tests. This proved to be the case even though evolutionary testing treats the software as black boxes, whereas developers are familiar with the function and structure of the software and achieve 100% branch coverage. An explanation might be the use of system calls, linkage, and compiler optimization whose effects on temporal behavior can only be guessed with difficulty by the developers. However, it should be noted that the execution times determined did not exceed the specified timing constraints for any of the tasks. The intensive testing has certainly strengthened the developers' confidence in a correct temporal behavior of the system. With an average number of 7 regression tests for each task, TESSY's entirely automated execution of regression testing has proved extremely useful.

Future work on testing real-time systems will focus on how static analysis techniques could support evolutionary testing, e.g. for search space reduction, to find a selection of evolutionary algorithms for test use, and to obtain information on internal states of the test object that may influence its temporal behavior. Further, the combination of evolutionary testing methods with static analysis techniques for the estimation of worst case execution times is meant to facilitate a precise forecast of the actual longest execution times of tasks [6]. Future plans, include the expansion of TESSY for integration testing and the examination of the suitability of evolutionary testing for system testing.

References:


[1] Wegener, J. and Pitschinetz, R. (1994): *TESSY - Yet Another Computer-Aided Software Testing Tool?* Proceedings of the European International Conference on Software Testing, Analysis & Review EuroSTAR '94, Bruxelles, Belgium.

[2] Grochtmann, M. and Grimm, K. (1993): *Classification Trees for Partition Testing*. Software Testing, Verification & Reliability, vol. 3, no. 2, pp. 63-82, Wiley.

[3] Grochtmann, M. and Wegener, J. (1995): *Test Case Design Using Classification Trees and the Classification-Tree Editor CTE*. Proceedings of the Software Quality Week '95, San Francisco, USA.

[4] Wegener, J. and Grochtmann, M. (1998): *Verifying Timing Constraints of Real-Time Systems by Means of Evolutionary Testing*. Real-Time Systems, vol. 15, no. 3, pp. 275-298, Kluwer Academic Publishers.

[5] Mueller, F. and Wegener, J. (1998): *A Comparison of Static Analysis and Evolutionary Testing for the Verification of Timing Constraints.* Proceedings of the IEEE Real-Time Technology and Applications Symposium RTAS '98, pp. 144-154, Denver, USA.

[6] Wegener, J., Pohlheim, H., and Sthamer, H. (1999*): Testing the Temporal Behavior of Real-Time Tasks using Extended Evolutionary Algorithms.* Proceedings of the European International Conference on Software Testing, Analysis & Review EuroSTAR '99, Barcelona, Spain.

# Automated Evaluation of COTS Components

**Carl J. Mueller, Bogdan Korel**

Computer Science Department
Illinois Institute of Technology
Chicago, IL 60616 USA
mueller@iit.edu, korel@iit.edu

**ABSTRACT**
Evaluating "Commercial-Off-The-Shelf" software is often a trying task. One factor that makes this task difficult is that source code for components is not available. In this paper, we present an automated approach for evaluating COTS software. This approach uses an interface probing strategy to automate the evaluation process. Evaluation begins with the developer providing a formal evaluation specification of the component. The evaluation specification is automatically translated into executable assertions that are used by an evaluation engine to generate automatically inputs to evaluate the component's behavior. When the evaluation engine generates an input causing an assertion violation, the component does not exhibit the expected behavior. On the other hand, when no assertion violation occurs, it gives the developer more confidence that the component exhibits the behavior as described in the evaluation specification. Our initial experience has shown that this approach may be a cost-effective way of evaluation of COTS components.

**Keywords**
automated testing, COTS component evaluation, black box testing, robustness testing, white box testing, automated test generation, formal methods

## 1 INTRODUCTION

Since the beginning of the 1990's, the demand for large and complex software systems has been steadily increasing. The development of these systems is difficult and costly. In recent years several new technologies have emerged that have a significant impact on new ways of software development. One of these technologies is the reuse and integration of previously developed software components into newly developing software systems. This approach has the potential to reduce cost and cycle time thus giving developers the ability to deliver a complex product faster and at a lower cost to a customer. Of particular interest is the ability to integrate specialized units of software called "commercial-off-the-shelf" (COTS) components. This allows developers to build software systems consisting of COTS components and software components developed in house.

Development with COTS components has many advantages [18]: functionality is instantly accessible, components may be less costly, and components may have been developed by experts in the area. Along with many advantages of using COTS components, there are several disadvantages. A developer is presented with a COTS component that often has only a brief description of its functionality, carries no guarantee of adequate testing, and has a limited description of overall component quality. Adding to these difficulties, the developer often does not have access to the source code of the component. Typically, COTS components are considered black boxes because developers only have access to their interfaces. The common way to interface with such components is through an application program interface (API). Developers wanting to use COTS components face the problem of determining the exact functionality and quality of these components. They want to have some assurances that the component's functionality corresponds to the expected functionality. In addition, developers want some assurances of component's quality; no one is willing to use a low quality component in, for example, a safety-critical software system. Another issue is portability of COTS components. In many cases, the expectation is that the behavior of a component(s) in one environment is the same as in another. Designers may want some assurance this expectation is correct.

Developers must have a good understanding of COTS components in order to integrate them properly into a system under development. From the limited information available about components, developers must identify component properties (e.g., functionality, limitations, pre-conditions) in order to identify which properties exhibited by the component are in conflict with expected properties, other components, or with a system design. Once the component is accepted for integration, these conflicts, or mismatches, must be repaired through component adaptation. Only by correcting these mismatches is it

possible to integrate the component into the system.

Component evaluation is one possible solution to these problems. Typically, a manual evaluation of COTS components consists of gathering information about the component's behavior from available documentation and performing interface probing. Interface probing is a technique where a developer designs a set of input cases, executes the component with these input cases, and analyzes the component's outputs. This process of probing helps the developer to evaluate component properties. After an initial evaluation, the developer may design additional input cases to clarify its functionality and limitations. This approach may be an effective way of evaluation of overall component functionality. However, one of the major disadvantages of this approach is that frequently a large number of input cases are needed to analyze a component. Some component properties can be easily evaluated by simple interface probing, but evaluation of other properties may require significant interface probing and may be very labor-intensive, tedious, and expensive. In addition, developers may frequently miss major component limitations and incorrectly assume certain component functionality that does not represent the actual component functionality. This may lead to incorrect use of the component when it is integrated with a software system under development. In summary, manual interface probing is a labor intensive and highly inaccurate approach.

## 2 AUTOMATED EVALUATION APPROACH

In this paper, we present an automated approach for evaluating COTS software. This approach uses an interface probing [12] strategy to automate this evaluation. In many cases, significant probing is necessary to evaluate whether a component has a specific behavior or not. Therefore, a major objective is automating the evaluation process. Evaluation begins with the developer providing a formal evaluation specification of the component that describes the expected component's behavior and the characteristics of the inputs. The evaluation specification may consist of assertions describing expected behavior and characteristics of component inputs. The assertion describing the component's behavior is called the post-condition assertion. The assertion describing the input is called the pre-condition assertion that describes the characteristics of input that yield the expected behavior. The evaluation specification is automatically translated into executable assertions. These executable assertions are used by an evaluation engine (an automated test case generator) that generates inputs to evaluate the component's behavior. When the evaluation engine generates an input causing an assertion violation, the component does not exhibit the expected behavior. On the other hand, when no assertion violation occurs, it gives the developer more confidence that the component exhibits the behavior as described in the evaluation specification.

It usually is not possible to define a complete set of assertions that describe the expected component's behavior at the beginning of the evaluation process. Using help files, vendor provided documentation and any other available source; it is possible to construct the assertions representing the evaluator's initial understanding of the component. An assertion violation occurs when the post-condition assertion evaluates as false. When analyzing the reason for each violation, the evaluator may determine that the assertions are inadequate. As a result, the evaluator may refine the evaluation specification.

Based on the component's interface and its formal evaluation specification the evaluation engine: (1) generates input(s), (2) executes the component with this input, and, (3) executes the code representing executable assertions. Each time an assertion violation is detected, the input and output are logged for the future review. These three steps are repeated until specified resources are exhausted, e.g., time limit, number of cases. If all the designated resources are exhausted and no violations or exceptions are detected, the developer may have more confidence that the component has the characteristics described in the evaluation specification.

Generating input data for interface probing is very similar to test case generation. As a result, existing automated test generation tools are used in the component evaluation process. Test data generation, for software, is the process of identifying a set of test cases satisfying a selected testing criterion. The existing automated test generation tools can be classified as random data generators, black-box test generators, white-box test generators, and robustness test generators. Each of these tools has its strengths and limitations depending on types of inputs and information available. As a result, the evaluation engine uses all four-test generation methods.

**Random data generator**

Random data generation [3] is the process of selecting, at random, component inputs and then executing the component on these inputs. Random inputs can be automatically generated based on a component interface specification for simple data types, e.g., integer, real, string, arrays, etc. However, for more advanced data types the developer must provide component specific routines to generate randomly values for input parameters or provide for each input parameter a set of values to be used in random generation.

**Black-box test generator**

Black box testing uses the program specification to design test cases. Two major black-box testing methods are widely used: equivalence partitioning and boundary-value analysis [2, 15]. In these methods program inputs fall into two categories: valid inputs and invalid inputs. In equivalence, partitioning input space is divided into valid partitions

(containing valid inputs) and invalid partitions (containing invalid inputs). This partitioning is done based on a program specification. Test cases are selected from each valid partition and each invalid partition. Boundary-value analysis considers boundaries between valid and invalid partitions and selects test cases on these boundaries and around boundaries. Several automated black-box test generation tools [1] are available for generating test cases from a test specification, e.g., pre-condition assertions. The evaluation engine uses pre-condition assertions to automate the process of black-box test generation using the same techniques employed in the existing black-box test generators.

**Robustness test generator**

A component is robust if it can function correctly despite invalid inputs, exceptional inputs, and stressful conditions. There exist a number of robustness test generators [13, 14, 17]. For example, Ballista [13] is a test generation tool that generates inputs having a high probability of causing an exception, e.g., a crash. Frequently, such test cases are dependent on a component's input data type. For example, for C pointers input values generated may include NULL and –1; for an integer data type input values generated may include 0, 1, -1, maximum integer value, maximum integer value minus 1, minimum integer value, minimum integer value plus 1. Our evaluation engine generates robust inputs similar to the inputs generated by the existing robustness test generators.

**White-box test generator**

White-box testing is the process of identifying a set of test cases that satisfies a selected structural testing criterion, e.g., statement coverage, branch coverage, etc. An automated white-box test generation tool generates input on which a selected element, e.g., a statement, is executed. A major problem of previously described generation methods is that they do not use information about the internal structure of an assertion to generate input violating the executable assertion(s). Although source code for the component is not available, source code for assertions is automatically generated and may be used during the evaluation [11]. Each assertion has a special source statement (target statement) whose execution indicates an assertion violation. The goal for the evaluation engine is to find the component input on which a target statement is executed. As a result, the existing methods of automated white-box test data generation are applicable.

When the source code of a COTS component is available, all of the existing test generation methods can be used, e.g., [4, 5, 6, 7, 8, 9, 10, 16]. However, in most cases the source code of a COTS component is not available. In such cases, only execution-oriented test generation methods may be used [7, 9, 10, 11]. Execution-oriented test generation starts by initially executing a component with arbitrary input. When a post-condition assertion is executed, its execution flow is monitored. During the assertion execution, the evaluation engine decides whether the execution should continue through the current branch or an alternative branch should be taken. For example, the currently executed branch does not lead to the execution of the target statement. If an undesirable execution flow at the current branch is observed, then a real-valued function is associated with the branch. Function minimization search algorithms are used to automatically find new input that will change the flow execution at this branch.

## 3 EXPERIMENT

The major goal of the experiment was to determine whether automated interface probing using a relatively simple formal evaluation specification could adequately evaluate commercial COTS components. To minimize subjectivity, we decided to evaluate the same commercial components existing in several different environments. These components were expected to provide the same behavior across all environments. As a result, the objective of the experiment was to use our approach to detect inconsistencies in these COTS components.

In our experiment, we selected several functions from the C standard library as COTS components. In the experiment, we used simple formal evaluation specifications expressed in terms of assertions. These assertions did not cover all aspects of component's functionality; they were relatively simple assertions capturing relationships between component's inputs and outputs. The starting point for the assertion development was the help facility of the target compiler, or other documentation. From these descriptions, initial assertions were developed and then modified as experience was gained with the component. These assertions were very easy to develop.

Automated interface probing was used for each component in different C++ environments. It was expected that standard C library functions (components) should behave consistently in all listed environments. In the experiment the following environments were used: Microsoft's Visual C/C++, Borland Builder 4, Borland Turbo C/C++, LINUX GNU C/C++, SGI GNU C/C++.

A component inconsistency is a situation where a component, with the same input, behaves differently across environments, i.e., a component produces different results on the same input in different environments. In the experiment, post-condition assertions were used to detect component inconsistencies. A component inconsistency is detected when a component with the same input produces a result that violates the assertion in one environment but it does not violate the assertion in another environment, or the component, with the same input, generates an exception condition in one environment and not in the other. In the latter case, no assertion is necessary because the exception condition is detected by the environment.

For each component under investigation, the same interface probing strategy was used in each C++ environment. During interface probing, all assertion violations and corresponding inputs were recorded. These inputs were then "applied" in the remaining environments to determine whether similar violations occurred. When no violation was observed in one environment, a component inconsistency was detected. When the same assertion violation occurred in all environments, the condition was not considered an inconsistency.

In the experiment, we investigated 8 components. We detected component inconsistencies in 6 components. In the following components inconsistencies were detected: atoi(), sin(), strncpy(), strcspn(), isdigit(), and isalnum(). In two components exception conditions occurred.

Our initial experience with the approach has shown that the effort required to develop formal evaluation specifications in the form of assertions pays off in detection of component inconsistencies. Using easy to develop evaluation specifications we were able to detect inconsistencies in commercial components.

## 4 CONCLUSIONS

In this paper, we presented an automated approach for evaluating COTS components. This approach uses interface probing to automate the evaluation process. Evaluation begins with the developer providing a formal evaluation specification of the component. The evaluation specification is automatically translated into executable assertions that are used by an evaluation engine to generate automatically inputs to evaluate the component's behavior. The evaluation engine uses the existing automated test generation methods to generate inputs during the evaluation process. An initial experiment has shown that the approach can effectively detect discrepancies in commercial components. The major advantage of the presented approach is that after a component's formal evaluation specification is provided the approach is fully automated.

The presented approach has been partially implemented. Currently we use a simple language to describe evaluation specifications. This language works fine for relatively simple components. We plan to develop a more powerful formal language that can be used to describe evaluation criteria of more complex COTS components. We also plan to perform a larger experiment to determine the effectiveness of the presented approach and to develop new methods of automated input generation that can be used in the evaluation process.

## 5 REFERENCES


[1] M. Balcer, W Hasling, T. Ostrand, "Automatic Generation of Test Scripts from Formal Test Specifications", Third Symp. on Testing, Analysis, and Verification, 1989, pp. 210-218.

[2] B. Beizer, Black-Box Testing. Techniques for Functional Testing of Software and Systems, Wiley & Sons, 1995.

[3] D. Bird, C. Munoz, "Automatic generation of random self-checking test cases," IBM Systems Journal, Vol. 22, No. 3, 1982, pp. 229-245.

[4] R. Boyer, B. Elspas, K. Levitt, "SELECT - A formal system for testing and debugging programs by symbolic execution," SIGPLAN Notices, Vol. 10, No. 6, 1975, pp. 234-245.

[5] L. Clarke, "A system to generate test data and symbolically execute programs," IEEE Tran. on Software Eng., Vol. 2, No. 3, 1976, pp. 215-222.

[6] R. DeMillo, A. Offutt, "Constraint-based automatic test data generation," IEEE Transactions on Software Eng., Vol. 17, No. 9, 1991, pp. 900-910.

[7] R. Ferguson, B. Korel, "The chaining approach for software test data generation," ACM Trans. on Soft. Eng. & Method., Vol. 5, No. 1, 1996, pp. 63-86.

[8] W. Howden, "Symbolic testing and the DISSECT symbolic evaluation system," IEEE Transactions on Software Eng., Vol. SE-3, No. 4, 1977, pp. 266-278.

[9] B. Korel, "Automated software test data generation," IEEE Transactions on Software Eng., Vol. 16, No. 8, 1990, pp. 870-879.

[10] B. Korel, "Dynamic method for software test data generation," Journal of Software Testing, Verification and Reliability, Vol. 2, 1992, pp. 203-213.

[11] B. Korel, A. Al-Yami. Assertion-Oriented Automated Test Data Generation. Proceedings of ICSE-18, 1996, pp. 71-80.

[12] B. Korel, "Black-Box Understanding of COTS Components," 7th International Workshop on Program Comprehension, 1999, pp. 92-99.

[13] N. Kropp, "Automatic Robustness Testing of Off-the-Shelf Software Components," CMU/ICES Tech. Rep. #01-27-98.

[14] B. Miller, et al., Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services, CSTR 1268, Univ. of Wisconsin-Madison, 1995.

[15] G. Myers, The Art of Software Testing, John Wiley & Sons, 1979.

[16] C. Ramamoorthy, S. Ho, W. Chen, "On the automated generation of program test data," IEEE Tran. on Software Eng., SE-2, No. 4, 1976, pp. 293-300.

[17] M. Schmid, F. Hill. "Data Generation Techniques for Automated Software Robustness Testing," The 16th Int. Conf. on Testing Computer Software (ICTCS'99), 1999.

[18] J. Voas, "COTS Software the Economical Choice?", IEEE Software, March/April 1998, pp. 16-19.

**Automating the testing of databases.**

**R A Davies*, R J A Beynon** and B F Jones*.**
***School of Computing, University of Glamorgan, Pontypridd, CF37 1DL, UK.**
****Now at Pontypridd Technical College, Pontypridd, UK**

Radavies@glam.ac.uk

## Abstract

We present a description of work in progress that investigates the feasibility of the automated production of test data to both populate a database for load testing and establishing the integrity of the stored data. A prototype test data generator has been produced that works on a simple ACCESS 97 database. This prototype performs the following basic functions: -

- database table structure analysis
- setting field entry limits by parsing validation rules
- valid & invalid test data generation
- error handling & recovery

## Introduction

The aim of this work is to generate test data automatically that checks

- the ability to store/retrieve, update/delete records correctly
- query testing
- the variation in performance with increased data and/or user numbers
- load testing
- the need to handle incorrect data (i.e. not obeying data integrity constraints)
- 'correctness' testing

Two of the functions that a fully automated test data generator has to provide are: -

- populate the database with valid data for load/query testing
- produce invalid data to test the constraints of the database by using limits/boundary conditions values of individual fields and referential relationships.

Current commercial database test data generators such as: -

- Test Byte 3
- Test Base
- DataTect

populate the database with random sets of characters/numbers dependent on the datatype of a given record field. The more sophisticated types allow users to define explicitly intrinsic and/or external data sets as sources for fields requiring names, cities etc to provide relevant entries and/or user definition of value ranges, permitted/forbidden values. Thus, as they require user input to set up field restrictions and knowledge of database structure, these fail to be fully automated. In addition, they fail to attempt to test the 'correctness' of the database by including the production and error handling of invalid data. A user-accessible log of any such errors produced will be of use in ensuring the database design satisfies user-requirements on individual field.

The prototype system developed works with Microsoft ACCESS. The software interrogates the database for its structure using the ACCESS table definitions; followed by parsing the Validation Rules obtained for individual fields to identify any present ranges, specified valid/invalid entry values etc. These values may then be stored and used in

data generation without the need for user input.

These conditions are then used to generate both valid records to populate the tables and records containing invalid values to test the validation of a given field. On entry the error would be 'trapped' and sent to an error log file and the entry program then needs to recover and move to next record (e.g. by replacing with a valid default entry or by deleting the erroneous record).

Since the database considered is comprised of a single table only, no testing of referential constraints is required, the prototype concentrates on testing of user/general constraints.

**Database 'Correctness'**

The following is a brief discussion of the concepts involved in the 'correctness' of databases. It is not intended to be fully comprehensive or mathematically rigorous.

'Correctness' is defined in the IEEE Standard Glossary of Software Engineering as freedom from faults, meeting of specified requirements and meeting user needs & expectations. [1]

In terms of databases this can be expressed as how successfully the database: -

1. models the real-life system – database design schema
2. the data store represents permitted 'real' possibilities only – i.e. data integrity

The former in relational databases is carried out within the Conceptual Schema Level, in which the system is designed in terms of Entities and their Relationships (i.e. in the E-R models etc.) [2].

The second is determined by ensuring data integrity constraints are handled. This can be done at either Internal or External Schema Levels, that is with the implementation of the database design itself (Internal) or as part of the application utilising the database (External) [2].

For the purposes of test data generation we will assume that the former method is used (Internal Schema Level) and the data integrity constraints are integral to the database implementation. This is likely to be the case since it is the safer method for ensuring data integrity [2].

Within Relational databases the data integrity constraints can be divided into three types: -

1. Key/Entity, to ensure that all records in a table are identified by a unique identifying attribute and there are no duplicates or null values.
2. Referential, if a record contains a foreign key field. The foreign key value must be a member of the set of values contained in the primary key field of the 'parent' entity or be null.
3. User/General and so relate to the data of a given field. They may be set to model the limits of the real-life condition the user needs to store information about.

[2]

From the above it can be seen that the automated 'correctness' testing suggested for the proposed data generator will relate to the data integrity constraints. However information on the database structure could be deduced from any error-log, providing an additional opportunity to check original design analysis.

In addition, for the limited scope of the current prototype (single isolated table),

it can be seen that the concentration is on the User/General constraints that are defined in the ACCESS Validation Rules since referential constraints are irrelevant.

Much of the current research in the area of database constraint testing is directed towards the problems of distributed databases [3,4 & 5]. In such databases, where data is stored across several individual dissimilar databases, it is necessary to reconcile several possibly heterogeneous components into a single whole. This involves the problems of: -

- generating a single global design schema from the constituent local schema
- combining multiple sets of integrity constraints into a single global set for application across the distributed system.

In terms of stand-alone databases, much of the current research is directed toward development of methodologies to draw up complete test-plans for testing database integrity [6]. That research forms a basis for the preparation of test data by independent means. However, the further step of including test data production itself is not considered.

**Method**

The separate processes carried out by the current prototype are listed below: -

1. interrogate the table definition to get its structure
2. store the structure
3. parse the validation rules for individual fields to produce field constraint data
4. use the results of the parsing to generate data to that will populate the database with a combination of valid and invalid records
5. store the generated test data in a text file
6. read the test data from the text file into the database
7. catch any error generated and write it to an error log and recover from the error

**Results**

The prototype was developed and tested with a simple table, 'Customers', comprised of 6 fields of representative data types (numeric, text, and date). In addition, the validation rules were written to contain the most commonly expected types of restriction on fields (upper/lower limits, permitted/non-permitted values, lists and entry patterns).

The Customer Table structure is outlined below: -

| Field Name | Type | Default |
|---|---|---|
| FirstName | Text | John |
| LastName | Text | Doe |
| Credit | Currency | £100 |
| Gender | Text | M |
| IDNo | Text | ZZZZ000ZZ |
| DoB | Date | Today's date |

The validation rules on the fields were as follows: -

| Field Name | Validation Rule |
|---|---|
| FirstName | >"A" |
| LastName | >="A" AND <="N" |
| Credit | <=5000 AND >=0 AND <>1000 |
| Gender | Can be like "M" or "F" but not like "X", "Y" or "Z" |
| IDNo | Can be like ZZ??###? |
| Dob | Between 31/12/1949 and 1/1/2000 |

The Parsing of the Validation Rules resulted in the following output: -

Field No 0

Field Name FirstName

Data Type No. 10
Validation Rule:[FirstName]>"A"
Rule Token # 0 [FirstName]
Rule Token # 1 >
Rule Token # 2 "A"
Range - Low A
Range - High
OK Values
Non Values
Value Pattern
**********
Field No 1
Field Name LastName
Data Type No. 10
Validation Rule:[LastName]>="A" And <="N"
Rule Token # 0 [LastName]
Rule Token # 1 >=
Rule Token # 2 "A"
Rule Token # 3 And
Rule Token # 4 <=
Rule Token # 5 "N"
Range - Low A
Range - High N
OK Values A N
Non Values
Value Pattern
**********
Field No 2
Field Name Credit
Data Type No. 5
Validation Rule:[credit]<=5000 And [credit]>=0 And <>1000
Rule Token # 0 [credit]
Rule Token # 1 <=
Rule Token # 2 5000
Rule Token # 3 And
Rule Token # 4 [credit]
Rule Token # 5 >=
Rule Token # 6 0
Rule Token # 7 And
Rule Token # 8 <>
Rule Token # 9 1000
Range - Low 0
Range - High 5000
OK Values 5000 0
Non Values 1000
Value Pattern
**********
Field No 3
Field Name Gender
Data Type No. 10
Validation Rule:Like "[MF]" And Like "[!XYZ]"
Rule Token # 0 Like
Rule Token # 1 "[MF]"
Rule Token # 2 And
Rule Token # 3 Like
Rule Token # 4 "[!XYZ]"
Range - Low
Range - High
OK Values M F
Non Values X Y Z
Value Pattern
**********
Field No 4
Field Name IDNo
Data Type No. 10
Validation Rule:Like "ZZ??###?"
Rule Token # 0 Like
Rule Token # 1 "ZZ??###?"
Range - Low
Range - High
OK Values
Non Values
Value Pattern ZZ??###?
**********
Field No 5
Field Name DoB
Data Type No. 8
Validation Rule:>#12/31/49# And <#1/1/2000#
Rule Token # 0 >
Rule Token # 1 #12/31/49#
Rule Token # 2 And
Rule Token # 3 <
Rule Token # 4 #1/1/2000#
Range - Low 12/31/49
Range - High 1/1/2000
OK Values
Non Values
Value Pattern
**********

## Examples of the valid data generated are: -

Nphhuat
Bkwujzw
460.698
M
ZZZM210K
18/02/68

Lalx
Abx
376.3512
M
ZZAP826Z
04/07/94

Examples of the invalid data generated are: -

A
Byr
1710.5961
F
ZZAK460S
27/01/64
Xuva
N*
1930.2896
F
ZZQG986A
30/03/92
Cteqkb
Jhmqtzy
-1
F
ZZUT699J
01/03/89

The valid and invalid data files were then merged into a single test data file. This file was then used as input to the database. A portion of the resulting error log file is given below: -

Table: Customers
Trapped Error # 3317
Error Description:-
One or more values are prohibited by the validation rule '|2' set for '|1'. Enter a value that the expression for this field can accept.
Record Number # 1
Field # 0 Name : FirstName
Invalid Entry : A
**********
Table: Customers
Trapped Error # 3317
Error Description:-
One or more values are prohibited by the validation rule '|2' set for '|1'. Enter a value that the expression for this field can accept.
Record Number # 3
Field # 1 Name : LastName
Invalid Entry : N*
**********
Table: Customers
Trapped Error # 3317
Error Description:-
One or more values are prohibited by the validation rule '|2' set for '|1'. Enter a value that the expression for this field can accept.
Record Number # 4
Field # 2 Name : Credit
Invalid Entry : -1
**********

## Restrictions/Limitations of Current Prototype

- It works on single isolated tables
- The validation rule condition is restricted to a value from its own field only
- Referential integrity is not checked as there are no foreign keys
- The available field data types are restricted to Numeric, Text & Date
- Only a single range of permitted values is allowed per field
- Only a single pattern format is allowed per field (e.g. ??## #?? For a Post Code)
- When an invalid data item is encountered when entering record, it is replaced with the default value in Table Definition – note problem if unique i.e. primary key.

## Conclusion

The current prototype demonstrates the feasibility of developing a fully automated test data generator for databases. It indicates that the basic operations of such an application (interrogation of database structure, parsing of validation rules etc.) are possible. This is shown by using a simple ACCESS 97 database consisting of a single table. Also it shows that the testing of the data integrity constraints defined in a database schema may be included in an automated data generation/table populating process.

As far as we know, both of the above functions are beyond the capabilities of currently available commercial data generators. This indicates that proposed full automation of data population and the inclusion of a degree of testing will reduce user input.

The areas for future development are as follows: -

- Include 'linked' tables to represent E-R design for a database to handle referential data constraints.
- Additional functionality – extra data types, use of defined data sets for types of field names e.g. names, towns etc.
- Make stand-alone – the present prototype is directly attached to the database in use.
- Automate the comparison between the predicted error log and the actual error log.

**References**


[1] IEEE Standard Glossary of Software Engineering Terminology,
'IEEE Software Engineering Standards Collection', IEEE, (1994).

[2] Stanczyk S, 'Theory & Practice of Relational Databases', Pitman, (1990).

[ 3] Gupta A et al., 'Efficient & Complete Tests for Database Integrity Constraint
Checking', Principles & Practice of Constraint Programming:
Ed. Borning A, 874, 173-80, (1994).

[4] Reddy MP et al., 'A Methodology for Integration of Heterogeneous Databases', IEEE Trans. On Knowledge & Data Eng. 6,920-33, (1994).

[5] Reddy MP et al., 'Formulating Global Integrity Constraints During Derivation of Global Schema', Data & Knowledge Engineering, 16, 241-68, (1995).

[6] Robbert MA,Maryanski TL, 'Automated Test Plan Generation for Database Application Systems', Proc. Of 1991 ACM Symposium on Small Systems, 100-106, (1991).

# Static Analysis

# ATGen: Automatic Test Data Generation using Constraint Logic Programming and Symbolic Execution

**Christophe Meudec**
Computing, Physics & Mathematics Department
Institute of Technology, Carlow
Kilkenny Road
Carlow, Ireland
+353 (0)503 70455
meudecc@itcarlow.ie

**ABSTRACT**
The verification and validation of software through dynamic testing is an area of software engineering where progress towards automation has been slow. In particular the automatic design and generation of test data remains, by in large, a manual activity. This is despite the high promises that the symbolic execution technique engendered when it was first proposed as a method for automatic test data generation.

In this work, we propose, and implement, a new approach based on constraint logic programming for the automatic generation of test data using symbolic execution.

After reviewing the symbolic execution technique, we present our approach for the resolution of the technical difficulties that have so far prevented symbolic execution from reaching its full potential. We then describe ATGen, our automatic test data generator, which is based on symbolic execution and uses constraint logic programming.

**Keywords**
Software Testing, Automatic Test Data Generation, Symbolic Execution, Constraint Logic Programming

## 1 INTRODUCTION

Developing software that is correct and behaves as expected is difficult. It is, at best, time consuming and expensive. To address these problems we propose a general approach using constraint logic programming and a tool for automating the design and generation of test data for software verification and validation.

Software verification involves checking that the software respects its specification. Software verification techniques include software inspections, formal proving of program correctness, static analysis of programs and testing.

Software validation involves checking that the software as implemented meets the expectations of the customer. It includes software reviews and acceptance testing where the software is exercised using tests provided by the customer. The customer may also want the software to be tested for particular circumstances for which tests have yet to be devised.

While we focus on testing for software verification and validation, we recognize that other techniques may be complementary in this area (in particular software inspections).

The testing phase can be supported by automatic tools. Three main categories of automation can be distinguished [34, 2]:

- Automation of administrative tasks, e.g. recording of test specifications and outcomes (useful for regression testing), test reports generation;
- Automation of mechanical tasks, e.g. the running and monitoring (for testing coverage analysis purposes) of the software under test within a given environment, capture/replay facilities allowing the automation of test suites execution;
- Automation of test generation tasks, i.e. the selection and the actual generation of test inputs;

While the first two areas are being well served by commercial tools—to a point that the expression '*automatic testing*' is often used as a synonym for automation of the tests execution only—the actual generation of test inputs is mostly still performed manually (with the exception of random testing).

It is in fact still the case that the automatic selection and generation of test inputs remains a challenge for tool developers [34].

This manual generation of test inputs implies that rigorous testing is laborious, time consuming, and costly. It also implies that rigorous testing is not actually widely applied.

The symbolic execution* technique, as proposed by King [27] more than 20 years ago, has the potential to help with the automation of the selection and generation of test inputs for a variety of problems. However, this potential has so far never been fully realized due to many technical problems [12].

* symbolic execution is also called symbolic evaluation

For completeness we acknowledge that new test data generation techniques with as wide a range of applications as symbolic execution have been investigated, e.g. [18, 39]. Other techniques, with a smaller focus, have also been proposed e.g. [30, 14].

It is our contention that our work places the automatic generation of test inputs for a variety of applications, as provided through symbolic execution, firmly in our grasp as demonstrated by our tool, ATGen.

After presenting the symbolic execution technique and its many potential applications, we review the traditional technical problems attached to it. We then give an overview of previous work in this area.

Our general approach for the resolution of the technical difficulties associated with symbolic execution is presented next. This general approach has been applied to a non-trivial test generation problem resulting in ATGen, our tool, which is presented and discussed before concluding.

## 2 SYMBOLIC EXECUTION

The symbolic execution of computer programs is an automatic static analysis technique that allows the derivation of symbolic expressions encapsulating the entire semantics of programs. It was first introduced by King [27] to help with the automatic generation of test data for dynamic software verification. As we shall see, verification is not the only important area where symbolic execution can be used.

### The Symbolic Execution Technique

Symbolic execution extracts information from the source code of programs by abstracting inputs and sub-program parameters as symbols rather than by using actual values as during actual program execution. For example, consider the following Ada procedure that implements the exchange of two integer variables:

```
procedure Swap(X, Y : in out integer)
is
  T : integer;
begin
  T := X;
  X := Y;
  Y := T;
end Swap;
```

After actual execution of, say, `Swap(5, 10)`, `X` will be equal to `10` and `Y` will be equal to `5`, i.e. the values of X and `Y` have been swapped. Actual execution provides a snapshot of the semantics of the source code.

Using symbolic execution captures exactly and entirely the semantics of the source code. This is performed by associating the assigned variables with a symbolic expression made up of input variables only. Here, we denote symbolic expressions by delimiting them using single quotation marks. In our example therefore, `T` is first assigned to `'X'`, `X` is then assigned to `'Y'` and finally, `Y` is assigned to the symbolic expression `'X'`.

Most programs are not simple sequential composition of assignments. In particular, the presence of a conditional statement, such as an `if`…`then`…`else`…, splits the execution of programs into different paths. In general therefore, symbolic execution records for each potential execution path, a traversal condition. This path traversal condition is the logical conjunction of the Boolean conditions encountered by the path. This condition must be satisfiable for the path to be feasible. Infeasible paths (i.e. paths which cannot be traversed because no input data exists which satisfies its path traversal condition) are not uncommon and cannot be ignored.

Consider the example below where `Max` is a global integer variable.

```
procedure Order(X, Y : in out integer)
is
begin
  if X > Y then
    Max := X;
  else
    Swap(X, Y);
    Max := X;
  end if;
end Order;
```

Symbolically executing the procedure `Order` we obtain two paths:

1. Path Traversal Condition: `'X > Y'`
   Path Actions: `Max = 'X'`
2. Path Traversal Condition: `'Not(X > Y)'`
   Path Actions: `Max = 'Y'`
   `X = 'Y'`
   `Y = 'X'`

A more advanced example is provided later. We do not review here, for lack of space, techniques for the actual implementation of symbolic execution. Rather the reader is referred to a comprehensive survey of implementation techniques [6]. As we shall see, the difficulties do not so much lie with the implementation of the symbolic execution technique per se but more with the exploitation of its potential.

### Exploitation of Symbolic Execution

The verification and validation of software are the main areas of applications for symbolic execution.

For completeness, we also mention that symbolic execution can help with the following:

- software debugging, re-engineering and comprehension [6] (e.g. by providing condensed information about program paths);

- software optimization, simplification and specialization [8, 28, 6] (e.g. by helping to identify loop invariants which can be moved out of iterative constructs, or by identifying unnecessary automatically inserted exception handling code [33]);
- applications to formal specifications can also be found [32, 29, 1];

*Software Verification*
We can distinguish four areas of interest:

- Automatic Test Data Generation for Coverage Testing

This is the first powerful usage of symbolic execution historically identified [27]. It can be extended to include data flow testing [5, 16].

Testing coverage criteria such as statement or decision coverage [5] have as their objective the execution of all statements or all decision outcomes, respectively, of the program under test. A symbolic executor can generate the path traversal condition of paths selected to achieve complete coverage. The path traversal conditions can then be sampled to obtain a set of test inputs which, by construction, achieves 100% (excluding unreachable code of course) coverage for the chosen testing criterion.

This application of symbolic execution requires the implementation of a path selection strategy, the ability to detect infeasible paths and the ability to sample satisfiable path traversal conditions to generate test inputs.

This capability would save a lot of manual effort as well as, typically, increase the level of overall coverage achieved.

- Automatic Test Data Generation for Path Domain Testing

Using coverage testing, a particular execution path is only tested once using a single test. It is often necessary however, to generate several tests for a single path in order to detect coincidental correctness [2, 12] or exercise the path using *'extreme'* values (as in boundary analysis [2]).

This can be achieved through analysis of the path actions (e.g. to detect the use of the remainder operator *'rem'* and generate a constraint to distinguish its usage from the modulo operator *'mod'*) or of the definition domain of variables and by adding constraints to the path traversal condition to force the generation of particular values.

This application requires the additional ability to generate constraints depending on the context of execution.

This extra testing has the potential to greatly increase the likelihood that errors will be detected in the program under test.

- Automatic Test Data Generation for Run-Time Errors Testing

Run-time errors occur when something unexpected occurs during the execution of a program (e.g. division by zero, access outside array bounds, variable overflow). They have the potential to crash the operating system.

There are two ways of dealing appropriately with run-time errors:

- Proving that the program is run-time error free;
- Inserting exception handling code to handle run-time errors;

The first approach is sometimes applied to safety critical software where it is acknowledged that preventing run-time errors is better than controlling their effects [1]. This approach falls within the remit of software proving.

Testing software that uses exception handling requires the generation of test inputs which will trigger the run-time error concerned. This can be achieved through specific path traversal conditions generation (e.g. to ensure that the denominator in a division takes zero for value) and sampling for test inputs generation.

To achieve this, the functional requirements for a testing tool are similar to the path domain testing application discussed previously.

Generating test inputs to trigger run-time errors is, of course, also necessary for programs that do not deal with run-time errors appropriately.

- Helping with Software Proving

Software proving is concerned with formal software verification. Symbolic execution is usually used to generate proof requirements involving a formal specification of the program under consideration [22]. Use of assertions for proving interesting properties of the software under consideration is also possible [1]. The proof requirements are then proved, or refuted, independently using a theorem prover. Theorem provers typically require human interventions. The same approach can be used to prove the absence of run-time errors [1].

This is the traditional role of symbolic execution during program proving.

Another angle is to attempt the generation of a test input negating the proof requirement [39]: if a test can be generated the proof need not be undertaken as it is bound to fail. Detecting instantly, at a low cost, that a proof will fail is attractive: commonly, many of the proof requirements attempted are unprovable and time-consuming to deal with.

This is applicable to proving that a program is run-time error free, as the first step should be to try to generate automatically a test triggering a run-time error.

*Software Validation*
Most of what we have discussed so far, under the software

verification heading, is applicable to software validation except that the tests generation requests would originate from the customer. The specific requirements of software validation however are often overlooked.

For example, it would be attractive to generate tests on a per scenario basis as proposed by the customer. Being able to answer reliably and quickly questions such as *'what happens if such and such variables have such and such values and this loop is taken 14 times?'* would be attractive. The customer may also wish to execute the software under consideration for everyday circumstances (e.g. avoiding extreme values) or in special operational modes (e.g. landing mode in a fly-by-wire software).

While the ability to generate and sample path traversal conditions would be required as before, a new requirement of our testing tool would also be necessary in our view. The obvious way of dealing with such test generation requests using symbolic execution would be through the judicious placing of assertions in the program source code. However, this is cumbersome for large programs. In our view, a higher level of usability, through the development of dedicated Graphical User Interfaces, is necessary to unlock the potential of test data generators for software validation purposes.

**Traditional Difficulties with Symbolic Execution**

Here we review the problems associated with symbolic execution in general. We can distinguish two distinct types of difficulties:

- Technical difficulties with the symbolic execution technique per se;
- Practical difficulties with the exploitation of the symbolic execution results;

*Technical Problems*

We can list in this category some features of programming languages that are challenging to deal with.

For example, array references can be problematic where the index is a not a constant but a variable—as is typically the case—as the particular array element referred to is then unknown. Symbolic execution can be performed in these cases with the generation of ambiguous array references in path traversal conditions [12]. The problem then is to decide the satisfiability of the conditions generated.

Loops are also difficult to deal with appropriately. Bounded loops can of course be unfolded as they do not create any new path in the program. Loops which are input variable dependent however, can be executed any number of times. Hence, there is the dilemma of the number of times the body of the loop should be traversed. Typically, symbolic executors generate path traversal conditions with loops executing zero, once or several times. This problem however should be dealt with according to the testing criteria under consideration and the feasibility or not of the current path.

Procedure and function calls can be handled by in-lining the sub-program code each time it is encountered or symbolically executing it once and using the results at each invocation [12].

Other characteristics of structured programming languages, which are difficult to deal with using symbolic execution, are dynamic memory allocation, pointers (especially pointer arithmetic as is allowed in the C programming language) and recursion.

Many of the technical problems faced by symbolic executors have been discussed by Coward in [12] and by Clarke and Richardson in [6].

It is our view that, although the generation of symbolic expressions along a given path in a program is not without technical difficulties, most of the restrictions usually imposed by symbolic executors on the source language that can be handled originate from the limitations of the techniques used for path feasibility analysis and test data generation [26] (i.e. practical problems associated with the exploitation of the results of the symbolic execution phase).

*Practical Problems*

Most symbolic executors simply generate all the syntactic paths in a program [1] (with special considerations for loops). It has been remarked by Coward in his review of symbolic execution systems [11], that this way of proceeding, besides wasting a lot of effort (because it is a purely syntactic process where feasibility of the intermediate paths is not checked during generation), may not be practical since a program may contain more paths that can reasonably be handled. Better, would be to integrate a path selection strategy within the symbolic executor to generate as few conditions as is necessary to achieve a particular testing criterion.

Further, and as we have seen, to exploit fully the potential of the symbolic execution technique it is necessary to be able to check the feasibility of the path traversal conditions generated and, for testing purposes at least, to be able to generate actual test data for feasible paths. Unfortunately, and as highlighted in the next section, the complexity of the path traversal conditions generated have, to date, proved too high to be tackled efficiently and automatically.

In our view, it is that fundamental problem that has hindered the wider use, and further development, of verification and validation tools based on symbolic execution rather than the perceived technical problems traditionally associated with the symbolic execution technique per se.

**Related Work**

Early research tackled the path feasibility problem using linear programming routines and rule-based checks [12, 11, 36, 7].

The problem with this approach is the inflexibility of the resulting tools. It may work well for conjunctions of linear conditions over integers, but separate techniques need to be used for, say, non-linear conditions over floating point numbers.

Furthermore, path traversal conditions typically are logical expressions over a mix of Boolean, integer, floating point number and enumeration variables organized in arrays and records: these cannot be solved using a single resolution strategy. At best, a lot of preprocessing needs to be performed before submitting subsets of a condition to a particular constraint resolution technique.

Syntactic simplification rules, while of value towards the representation of traversal conditions in a simplified form [1], are unlikely to detect many infeasible paths as such.

Using a theorem prover, as illustrated in [26, 19], may allow the handling of arrays where the index is not a constant. However, while using axiomatic rules for proving that a particular set of symbolic expressions over arrays is unsatisfiable may be suitable, it cannot be applied to linear expressions over, say, integers. Also, on their own, theorem proving tools are not suited for generating test data satisfying a particular path traversal condition.

So, while many separate techniques have been employed in previous attempts at determining path feasibility, the sheer complexity of most path traversal conditions has meant that, in practice, the underlying language on which symbolic execution is applied must be simplified and that the complexity of the path traversal conditions must be low for the approach to succeed (e.g. linear expressions over either integer or floating point variables but not mixed conditions where floating point and integer variables are used).

Therefore, the source language typically handled by testing tools based on symbolic execution is a small subset of its original [12] and no test data generation facility is provided: the tool only performs path feasibility analysis on all the syntactic paths [19].

## 3 OUR APPROACH

Our underlying approach centers on the tighter integration of the different sub-systems making up a test data generator based on symbolic execution, by using a constraint logic programming language. We have two overiding concerns: reducing the amount of wasted effort during generation of the symbolic expressions and enlarging the typical programming language subset that can be efficiently tackled by symbolic execution. Coincidentally, we are, in effect, taking further the general ideas presented by Hamlet in [21] for the rapid implementation of general testing tools.

### Closer Integration

To avoid the generation of many paths with unsatisfiable path traversal conditions, and of paths which are not required for the fulfillment of the testing criteria under consideration, it is necessary to integrate the following, traditionally separate, elements of a test data generator:

- Symbolic executor;
- Path selector;
- Path feasibility analyzer;

Doing so would make it possible to check, during their symbolic execution, the feasibility of required paths only. Thus, we would avoid the heavy overhead engendered by the generation of unnecessary or infeasible paths.

While this approach is not a new proposal [12], its successful implementation has, to date, been elusive.

Additionally, we must also provide the means for the automatic sampling of satisfiable path traversal conditions so as to generate actual test data for the, known feasible, selected paths.

Constraint logic programming is the paradigm that has allowed us to realize these aspirations.

### Use of Constraint Logic Programming

As we have seen, we want to check the satisfiability of algebraic expressions. I.e. given an algebraic expression, along with the variables involved and their respective domains, we must show that there exists an instantiation of the variables which reduces the expression to true. In effect, an algebraic expression constrains its variables to a particular set of values from their respective domains. If any of the sets are empty, the assertion is reduced to false and is said to be unsatisfiable. Thus, an algebraic expression is a system of constraints over its variables.

Hence, we have a Constraint Satisfaction Problem (CSP): we want to search the variable domains for solutions to a fixed finite set of constraints.

*Constraint Satisfaction Problems (CSPs)*

CSPs (see [17] for an informal introduction) are in general NP complete and a simple '*generate and test*' strategy, where a solution candidate is first generated then tested against the system of constraints for consistency, is not feasible. Constraint satisfaction problems have long been researched in artificial intelligence and many heuristics for efficient search techniques have been found. For example, linear rational constraints can be solved using the well-known simplex method [13].

To implement the kind of solver we require, e.g. able to work with non-linear constraints over floating point numbers and integers, we could implement these heuristics by writing a specialized program in a procedural language (such as C, or using an existing solving routines library). Nevertheless, although the heuristics are readily available, this approach would still require a substantial amount of

effort and the resulting program would be hard to maintain, modify and extend. Ideally, we would like to concentrate on the '*what*' rather than the '*how*', i.e. we are more interested in the problem of combining the heuristics rather than in implementing the internal mechanism of each individual heuristic search technique.

The advantages of logic programming, mainly under the form of the Prolog programming language [4], over procedural programming have long been recognized [21]: the '*what*' and the '*how*' are more easily separated since Prolog is based on first order predicate logic and has an in-built resolution computation mechanism. However, Prolog's relatively poor efficiency when compared to procedural languages has hindered its general acceptance.

For CSPs, however, Prolog is still the language of choice. Searches are facilitated by its in-built depth-first search procedure and its backtracking facilities. However, even in this area Prolog suffers from a general lack of facilities to express complex relationships between objects (terms): the semantics of objects has to be explicitly coded into a term. This is the cause of the perceived poor mathematical handling capabilities of Prolog when compared with its other facilities: only instantiated mathematics can be dealt with readily. Further, the basic in-built depth-first strategy tends to lead to a '*generate and test*' approach to most problems: specialized heuristics must be implemented to prune the search space.

*Constraint Logic Programming*

Constraint Logic Programming (CLP), as introduced by Jaffar and Lassez [25], reviewed by Colmerauer [10] and discussed in [9], alleviates these shortfalls by providing richer data structures on which constraints can be expressed and by using constraint resolution mechanisms (also known as decision procedures) to reduce the search space. When the decision procedure is incomplete—e.g. for non-linear arithmetic constraints—the problematic constraints are suspended, we also say delayed, until they become linear. Non-linear arithmetic constraints can become linear whenever a variable becomes instantiated (or bound). This can happen when other constraints are added to the system of constraints already considered or during labeling.

The labeling mechanisms further constrain the system of constraints according to some strategy. It can be viewed as a process to make assumptions about the system of constraints under consideration. It is a very powerful mechanism and it is used to awaken delayed constraints or to generate a solution to an already known satisfiable system of constraints.

To deal with non-linear problems, the labeling strategies used are critical to the overall efficiency of the solver. A discussion of constraint satisfaction using CLP can be found in [24].

We now give an example of constraint resolution involving integers.

In Prolog the equality:

```
X*X + Y = 10
```

results in failure, since in Prolog equality only holds between syntactically identical terms and `X` is just a variable of no particular type. Using a CLP language however, it is possible to code the semantics of `X` and `Y` as being integer-like and constrain them such that `X*X + Y = 10` holds. The constraint resolution mechanism will detect the constraint as non-linear and reduce it as follows:

```
X = X, Y = Y
X*X + Y = 10 is delayed
```

I.e. the system of constraints is satisfiable (subject to consideration of the delayed constraints) and a simplified version is internally held. A labeling strategy must impose further constraints on either `X` or `Y` for the satisfiability of the system of constraints to be confirmed.

During labeling, a not so efficient strategy would select `Y` to be sampled first. The sampling strategy would then instantiate `Y` with, say, `2` thus awaking and simplifying the delayed constraint to `X*X = 8`. This constraint would have to be delayed. The labeling strategy now attempts to instantiate `X` repeatedly without success (because failure in this case actually occurs on the entire definition domain of `X`) which induces backtracking in the traditional, logic programming, manner. Eventually `Y` is instantiated to another value, say `1`, thus reducing the constraint store to:

```
X = X, Y = 1
X*X = 9 is delayed
```

The labeling mechanism now attempts to awake the delayed constraint: this can only be achieved through instantiating `X`. Eventually `X` will be instantiated with `-3` or `3` and the system of constraints will be declared satisfiable and the sample, say, `X = 3, Y = 1` will also be available. To succeed, this labeling strategy has generated thousands (if the initial domain of the variables is of that order) of futile assumptions.

A more efficient labeling strategy would recognize that `X` is the variable on which the linearity of the delayed constraint depends and attempt to constrain it first. Any value in the domain of `X` will make the delayed constraint linear thus allowing the constraint resolution mechanism of the underlying solver to detect satisfiability directly. E.g. `X = -5` will awaken the delayed constraint and the solver would directly yield:

```
X = -5, Y = -15
```

This latest labeling strategy is adequate as only one assumption is made to yield a positive outcome.

It is this general approach that we have customized to be applicable to path traversal conditions as generated during symbolic execution.

CLP languages are ideal for our purpose as their in-built resolution mechanism removes most of the needed development effort and still offer the flexibility of logic programming. In fact, they allow the rapid development of efficient, dedicated, constraints solvers.

## 4 ATGen: AN AUTOMATIC TEST DATA GENERATOR

ATGen is our prototype testing tool implemented using the underlying approach outlined in the previous section.

The particular constraint logic programming environment used to implement ATGen is ECLiPSe [15]. ECLiPSe is a Prolog based system that serves as a platform for integrating various logic programming extensions, in particular CLP. ECLiPSe is distributed with many valuable libraries implementing various constraint solvers (over integers, rational numbers, sets etc.). Other similar environments may well be as equally suited.

### Current Area of Application

While it should be clear that our approach is general enough to be applied to many structured programming languages for a variety of purposes we have chosen the initial area of application to be as compelling as possible.

Hence, the current area of application of ATGen is the automatic generation of test data to achieve 100% decision coverage of SPARK Ada programs.

*Decision Testing*

In decision testing [5, 2] the aim is to test all decision outcomes in the program. Typical decisions are Boolean expressions controlling the flow of execution in the program such as in conditional constructs and loops. Discounting infeasible decision outcomes [5] we aim to generate a test data suite achieving 100% decision coverage.

Note that the current implementation of our path selection strategy is not aimed towards producing the smallest test set possible but towards the fastest generation of a set of tests achieving coverage. Thus, some redundant paths may well be generated.

*SPARK Ada*

SPARK Ada [1] is a subset of the Ada programming language designed in particular for the development of high integrity software. It is the most popular Ada subset for safety critical software.

Briefly, the following Ada features are excluded from SPARK Ada: concurrency, dynamic memory allocations, pointers, recursion, and interrupts. The reader is referred to Barnes [1] for a complete definition of SPARK Ada.

Formally SPARK Ada is not just a subset of Ada, as it also requires additional annotations to give extra information about the program. This extra information can then be handled by SPARK analysis Tools (such as the SPARK Examiner [1]) to perform various static program analysis tasks (such as data flow analysis [1]).

We however discard any SPARK annotations and only consider the Ada constructs for our test data generation purposes (ATGen however could easily be adapted to handle FDL—Functional Description Language—constructs making up the outputs of the SPARK analysis tools [1]).

ATGen handles the entire SPARK Ada subset including Boolean, integer, floating point (represented using infinite precision rational numbers), enumeration types, records, multi-dimensional arrays, all loop constructs, functions and procedures calls. Further, there are no technical reasons why we may not extend the Ada subset currently handled by ATGen beyond SPARK.

### Overall Structure

ATGen is composed of a pre-parser written in C and of roughly 4200 lines of commented Prolog code divided in seven modules.

The pre-parser transforms* the SPARK code into a list of Prolog facts. This transformation is purely syntactical (e.g. the first letter of variables is put into upper case, labels for conditions are automatically generated). For interest, we give below the parsed version of the second loop of `quotient`, our next example.

```
while(cond(C2, T <> D), stmts([
  assign(Q, Q * 2),
  assign(T, T / 2),
  if(cond(C3, T <= R), then(stmts([
    assign(R, R - T),
    assign(Q, Q + 1)
  ])),
  elsifs([]), else(stmts([])))
]))
```

This intermediate representation of the SPARK source code is compiled into ATGen as an ordinary Prolog program.

The symbolic execution of the program under consideration is directed according to the testing coverage criteria chosen and the feasibility analysis of the current subpath: infeasible or redundant subpaths are immediately abandoned and the system backtracks in an ordinary Prolog manner. The path traversal condition of suitable paths is sampled to generate test data. This entire process is repeated, through backtracking, until the testing coverage

* this transformation is still partially performed manually, but a parser is nearing completion using a YACC-like parser generator.

criteria is fulfilled.

Below we give a rough estimate of where the development effort was spent.

- 30% solving activities. Mostly concerned with customization and extension of the solving capabilities provided with ECLiPSe;
- 30% symbolic execution per se. Dealing with sub-program calls, iterative and conditional constructs, assignments;
- 20% source language features manipulation. Mainly concerned with the data structures of the source language (arrays, record, enumeration types);
- 10% labeling. Implementing overall and data type specific labeling strategies;
- 5% path selection strategy implementation;
- 5% test report generation. In particular printing of arrays, output domains;

The reader can infer from the above what effort would be involved in extending ATGen (for another language, a new coverage measure, improved labeling strategies etc.)

**Characteristics and Limitations**
We list below some interesting aspects of ATGen.

- The tests generated for coverage testing are designed not to generate run-time errors (including avoiding internal overflow of expressions);
- Using ATGen, actual test execution becomes unnecessary since the actual test output is provided along side the test inputs. However, it may still be necessary, to comply with the independent verification requirement of safety critical systems for example, to actually execute the test generated;
- Annotation of the SPARK source code is not needed;
- ATGen can be used for integration testing purposes [23] which is maybe the area where the manual design of test data is the most difficult;
- ATGen itself need not be of high integrity: the actual level of code coverage achieved can be checked using a third party tool;
- Sometimes path feasibility will be too time consuming to infer (mainly when the path traversal condition involves complex non-linear relations between floating point variables). Better heuristics for labeling and advances in CLP languages in general should reduce this problem in the future. Currently ATGen in such situations issues a warning message indicating which path has been considered infeasible by default;

**Example**
We reproduce, verbatim, an example given in [18] to demonstrate the problems associated with symbolic execution:

```
procedure quotient(n: Some_Integer,
                   d: Some_Integer) is
-- calculate quotient and
-- remainder of the integer
-- division of n by d, (n>0, d>0)
  q: Some_Integer := 0;
  r: Some_Integer := n;
  t: Some_Integer := d;
begin
  while r >= t loop           --C1
    t := t * 2;
  end loop;
  while t /= d loop           --C2
    q := q * 2;
    t := t / 2;
    if t <= r then            --C3
      r := r - t;
      q := q + 1;
    end if;
  end loop;
-- manipulate r and q;
end quotient;
```

The difficulties with `quotient` are that both loops are input variable dependent and that the second loop must be executed exactly the same number of times as the first loop was for the path under consideration to be actually feasible. Further, the path traversal conditions generated involve non-linear arithmetic.

A typical ATGen output for the procedure `quotient` using the decision coverage testing criteria is given below.

Path: `C1 false, C2 false`
Test Data: `D = 10084, N = -20016`
Test Result: `T = 10084, R = -20016, Q = 0`

Path: `C1 true, C1 true, C1 false, C2 true, C3 true, C2 true, C3 false, C2, false`
Test Data: `D = 5836, N = 12905`
Test Result: `T = 5836, R = 1233, Q = 2`

We make several remarks about this result:

- The above result is generated in under 1.5 seconds on average using a 450MHz Pentium III based machine;
- ATGen is non-deterministic: the actual paths and test data generated may differ on subsequent runs;
- More information, than we have space here for, per path is available (such as the actual path traversal condition);
- The second path generated actually makes the first one

redundant for decision coverage purposes;

- The path traversal condition for the second path is: `N >= D and N >= D*2 and not(N >= D*2*2) and D*2*2 /= D and D*2*2/2 <= N and D*2*2/2 <> D and not(D*2*2/2/2 <= N - D*2*2/2) and not(D*2*2/2/2 <> D)`

## 5 FUTURE WORK

Below are our plans for future work on ATGen.

**Demonstrating Practicality***

We must better demonstrate the practicality of ATGen for real world testing applications. Therefore, while we need to evaluate ATGen using automatically generated code (as in [18]), our main motivation should be to seek actively real world software testing problems in the best engineering research tradition [35].

**Increasing Usability**

If the potential of symbolic execution is to be realized, ATGen must provide better ways for the vast amount of information that can be generated to be channeled efficiently to the human testers. Further, the execution of the test data generator should be easy to parameterize to facilitate software validation activities.

We believe that this requires the development of a Graphical User Interface (GUI) with facilities to visualize and manipulate the control flow organization of the software under consideration. We have started work on this aspect.

We remark that program analysis tools in general, need to allow greater interaction with the user to expand successfully in a software engineering environment [31].

**Increasing Versatility**

We would like to increase the number of test coverage criteria handled by ATGen (in particular MCDC [5], Modified Condition Decision Coverage, which is required for airborne systems [38]), and expand into data flow testing [5, 16].

Larger subsets of Ada than SPARK may also be considered as well as other languages (e.g. Java [20] or C).

In addition, the application of ATGen in the area of run-time errors testing deserves further investigations.

**Increasing Performance**

While we have been pleasantly surprised by the overall performance of ATGen in terms of execution time and capability at detecting immediately infeasible paths, we recognize that further improvements may well be necessary. In particular we need to investigate better labeling strategies including moved based heuristics such as Hill climbing, Simulated Annealing and Tabu search [15, 37].

## 6 SUMMARY

ATGen automatically generates test data for total decision coverage of SPARK Ada programs.

It implements our general approach for solving the traditional problems associated with the symbolic execution technique.

Our approach is centered on tighter integration of the various components making up a test data generator using constraint logic programming. This use of constraint logic programming is, to our knowledge, unique to our work.

We have presented our plans for future work and are confident that ATGen will be successfully applied on real world software testing problems in the near future.

**ACKNOWLEDGEMENTS**

Some preliminary results, concerning the work presented, were obtained while the author was employed by the University of York, UK.

**REFERENCES**

1. Barnes, J. *High Integrity Ada: The SPARK Approach*, Addison-Wesley, ISBN 0-201-17517-7, 1997.
2. Beizer, B. *Software Testing Techniques*, 2nd Edition, Van Nostrand Reinhold, ISBN 0-442-20672-0, 1990.
3. Bicevskis, J., Borzovs, J., Straujums U., Zarins A., and Miller, E.F. SMOTL—A System to Construct Samples for Data Processing Program Debugging, *IEEE Trans. Software Eng.* 15:60-66, 1979.
4. Bratko, I., *Prolog Programming for Artificial Intelligence*, 2nd Edition, Addison-Wesley, ISBN 0-201-41606-9, 1990.
5. British Computer Society Specialist Interest Group in Software Testing (BCS SIGIST), *Standard for Software Component Testing*, Working Draft 3.3, 1997.
6. Clarke, L.A., and Richardson, D.J. Applications of Symbolic Evaluation, *J. Systems Software*, 5:15-35, 1985.
7. Clarke, L.A., Richardson, D.J., and Zeil, S.J. Team: A Support Environment for Testing, Evaluation and Analysis, In *Proceedings Software Engineering Symposium of Practical Software Development*, pp. 153-162, Nov. 1988.
8. Coen-Porisini, A., De Paoli, F., Ghezzi, C., and Mandrioli, D. Software Specialization Via Symbolic Execution, *IEEE Trans. Software Eng.*, 17(9):884-899, Sep. 1991.
9. Cohen, J. Constraint Logic Programming Languages, *Commun. ACM*, 33(7):52-68, Jul. 1990.

* It is hoped that further results will be available in time for the workshop as well as a tool prototype.

10. Colmerauer, A. An Introduction to Prolog III, *Commun. ACM*, 33(7):69-90, Jul. 1990.

11. Coward, P.D. Symbolic Execution Systems—A Review, *Software Engineering Journal*, 3(6):229-239, Nov. 1988.

12. Coward, P.D. Symbolic Execution and Testing, *Information and Software Technology*, 33(1):53-64, Jan./Feb. 1991.

13. Dantzig, G.B. *Linear Programming and Extensions*, Princeton University Press, Princeton, New Jersey, 1963.

14. Demillo, R., and Offutt, A. Constraint-Based Automatic Test Data Generation. *IEEE Trans. Software Eng.*, 17(9):900-910, 1991.

15. ECLiPSe Release 4.2, Imperial College London, 1999, *http://www.icparc.ic.ac.uk/eclipse/*

16. Frankl, P.G., and Weyuker, E.J. An Applicable Family of Data Flow Testing Criteria, *IEEE Trans. Software Eng.*, 14(10):1483-1498, Oct. 1988.

17. Freuder, E. The Many Paths to Satisfaction, In *Proceedings ECAI'94 Workshop on Constraint Processing*, Amsterdam, Aug. 1994.

18. Gallagher, M.J., and Narasimhan, V.L. ADTEST: A Test Data Generation Suite for Ada Software Systems, *IEEE Trans. Software Eng.*, 23(8):473-484, Aug. 1997.

19. Goldberg, A., Wang, T.C., and Zimmerman, D. Applications of Feasible Path Analysis to Program Testing, In *Proceedings ISSTA'94* (Seattle, WA, Aug. 1994)

20. Gosling, J., Joy, B., and Steele, G. *The Java Language Specification*, Technical Report, Sun Microsystems, Aug. 1996.

21. Hamlet, D. Implementing Prototype Testing Tools, *Software Practice and Experience*, 25(4):347-371, Apr. 1995.

22. Hantler, S.L., and King, J.C. An Introduction to Proving the Correctness of Programs, *ACM Computing Surveys*, 8(3), pp. 331-353, September 1976.

23. Harrold, M.J., and Soffa, M.L. Selecting and Using Data for Integration Testing, *IEEE Software*, 58-65, Mar. 1991.

24. Hentenryck, P.V. Constraint Satisfaction using Constraint Logic Programming, *Artificial Intelligence*, 58:113-159, 1992.

25. Jaffar, J., and Lassez, J-L. Constraint Logic Programming, In *Proceedings POPL'87*, pp. 111-119, Munich, Jan. 1987, ACM Press.

26. Jasper, R., Brennan, M., Williamson, K., Currier, B., and Zimmerman, D. Test Data Generation and Feasible Path Analysis, In *Proceedings ISSTA'94* (Seattle, WA, Aug. 1994) 95-107.

27. King, J.C. Symbolic Execution and Program Testing, *Commun. ACM*, 19(7):385-394, 1976.

28. King, J.C. Program Reduction Using Symbolic Execution, *SIGSOFT Software Engineering Notes*, 6(1):9-14, 1981.

29. Kneuper, R. Symbolic Execution: a Semantic Approach, *Science of Computer Programming*, 16(3), pp. 207-249, Oct. 1991.

30. Korel, B. Automated Test Data Generation for Programs with Procedures, In *Proceedings ISSTA'96*, pp. 209-215.

31. Le Métayer, D. Program Analysis for Software Engineering: New Applications, New Requirements, New Tools, *ACM SIGPLAN Notices*, 32(1):86-88, Jan. 1997.

32. Meudec, C. Tests Derivation from Model Based Formal Specifications, In *Proceedings 3rd Irish Workshop in Formal Methods*, Galway, Jul. 1999, http://www.ewic.org.uk/ewic/.

33. Muller, G., and Schultz, U.P. Harissa: A Hybrid Approach to Java Execution, *IEEE Software*, 44-51, Mar./Apr. 1999.

34. Ould, M.A. Testing—A Challenge to Method and Tool Developers, *Software Engineering Journal*, 6(2):59-91, Mar. 1991.

35. Parnas, D.L. On ICSE's "Most Influential Papers", *ACM Software Eng. Notes*, 20(3), 1995.

36. Ramamoorthy, C.V., Siu-Bun, F.H., and Chen, W.T. On the Automated Generation of Program Test Data, *IEEE Trans. Software Eng.*, 2(4):293-300, Dec. 1976.

37. Rayward-Smith, V.J., and al. Editors. *Modern Heuristic Search Methods*, Wiley, 1996.

38. RTCA, *Software Considerations in Airborne Systems and Equipment Certification Guideline*, Radio Technical Commission for Aeronautics, Number DO-178A, Mar. 1985.

39. Tracey, N., Clark, J., and Mander, K. Automated Program Flaw Finding using Simulated Annealing. In *Proceedings ISSTA'98*, pp. 73-81.

# Program Analysis and Test Hypotheses Complement

R. M. Hierons and M. Harman

April 18, 2000

## Abstract

This paper considers ways in which program analysis and test hypotheses complement, focusing on one particular example: the uniformity hypothesis. Conditioned slicing can be used to either provide confidence in the uniformity hypothesis, identify faults, or suggesting refinements to the hypothesis. The existence of a uniformity hypothesis assists in the production of small conditioned slices which might then be analysed further. keywords: Program verification, test hypotheses, the uniformity hypothesis, program analysis, conditioned program slicing.

## 1 Introduction

Most approaches to program verification can be categorised as one of dynamic testing and program analysis. Dynamic testing involves exploring the behaviour of the *implementation under test (IUT)* when given particular input values. Within the verification context, program analysis involves studying the source code of the IUT in order to derive information that might either increase the confidence in the correctness of the IUT or detect faults in the IUT.

In general, it is not possible to produce a finite test set that is guaranteed to determine correctness. There are, however, techniques that generate tests that are guaranteed to determine correctness as long as the IUT satisfies certain conditions. These conditions have been called test hypotheses ([11]) and design for test conditions ([19]). Section 2 discusses test hypotheses. Testing might then be seen as a process of choosing an appropriate set of test hypotheses and then generating a corresponding test set.

If the hypotheses do not hold, the corresponding test set may be ineffective and inefficient. Thus, it is important to use test hypotheses that hold.

Previous work has largely focussed on introducing new hypotheses and generating tests in the presence of hypotheses ([12, 8, 31, 32, 9, 27, 5, 11, 17]). This paper instead concentrates upon semi-automated techniques for establishing that such hypotheses do hold. Specifically the relationship between the uniformity hypothesis and program analysis, through the use of conditioned slicing, is described. Slicing shall be briefly reviewed in Section 3.

Sections 4 and 5 will discuss the following ways in which this relationship may be used.

1. An instance of the uniformity hypothesis, which represents expert knowledge about the IUT, might be used to simplify program analysis.

2. Program analysis might be used to either provide confidence in, refute or refine proposed test hypotheses.

It will thus be demonstrated that there exists a symbiotic relationship between program analysis and test hypotheses.

## 2 Test hypotheses

Suppose $I$ is to be tested against a specification $M$ with input domain $D$. It is normal to assume that $I$ accepts the

same class of inputs as $M$, though an error may result for some of these input values. Without any further knowledge about $I$ there is, in general, no finite test set that determines correctness.

Fortunately this does not represent the normal scenario in testing. The tester has some expert knowledge about $I$ and $I$ is not, in general, merely a black box: it is often possible to examine the code used to produce $I$. There is thus further information about $I$ that may be utilised in test generation. This information might be expressed as properties of $I$ called *test hypotheses*.

Suppose that $F$ denotes the set of possible behaviours of the IUT. $F$ is often called a *Fault Model* ([22]). Suppose, further, that the current set of hypotheses is $H$ and $F_H$ denotes the set of behaviours, from $F$, that are consistent with $H$. Let $I' \leq M$ denote that $I'$ conforms to $M$ and $I' \leq_T M$ denote that the $I'$ conforms to $M$ on $T \subseteq D$. Naturally, the notion of conformance used depends upon the specification language. The following defines what it means for a test set $T$ to be guaranteed to determine correctness under $H$.

**Definition 1** *Test set $T$ is* complete *with respect to $H$ if and only if* $\forall I' \in F_H . I' \leq M \iff I' \leq_T M$.

Two related notions, of a test being unbiased and valid, have been described ([11]). A test is *valid* if it rejects all faulty implementations that satisfy the test hypotheses. A test is *unbiased* if it cannot reject a conforming implementation that satisfies the test hypotheses. Then a test is complete if and only if it is unbiased and valid.

Clearly, the exhaustive test set $D$ is complete with respect to every hypothesis $H$. Exhaustive testing is, however, rarely practical. Given $M$ and $I$, there are the following, inter-related, challenges.

1. To devise some set $H$ of test hypotheses, that $I$ is likely to satisfy, such that there is a corresponding feasible complete test set.
2. To determine whether $I$ satisfies $H$.
3. To generate a complete test set for $I$ with respect to $H$.

The development of an appropriate set of hypotheses can proceed via refinement ([11]). Some minimal hypothesis is produced and this is refined through a number of steps. The minimal hypothesis might, for example, be that $I$ is equivalent to some unknown element from fault model $F$ or simply that the input and output domains for $I$ are the same as those for $M$. Each refinement strengthens the test hypotheses and thus, potentially, allows a smaller complete test set.

Many test generation techniques are based around partitioning the input domain $D$ into a finite set $D^M = \{D_1, \ldots, D_k\}$ of subdomains such that, according to $M$, all elements in a subdomain should be processed in the same way ([12, 31, 32, 9, 27, 11, 17]). The *uniformity hypothesis* says that if the input of one value in some $D_i \in D^M$ leads to a failure then all values in $D_i$ lead to failures.

It is to be expected that there is some (unknown) partition $D^I$, of the input domain, such that the behaviour of $I$ is uniform on each subdomain of $D^I$. The uniformity hypothesis is thus based upon the assumption that $D^M$ and $D^I$ are similar. If the uniformity hypothesis holds it is sufficient to choose one value from each $D_i \in D^M$. However, the test for some $D_i \in D^M$ is normally complemented by tests around the boundaries of $D_i$ ([32, 9]) which are expected to find any small errors in the boundaries.

It has been noted ([29]) that if the partitions $D^M$ and $D^I$ were known, the behaviours of $I$ and $M$ could be compared on each subdomain from $D^{IM} = \{D_i \cap D_j \mid D_i \in D^M, D_j \in D^I\}$. While we do not consider the generation of $D^I$, this idea from ([29]) provided the inspiration for much of the work contained in this paper.

# 3 Program slicing and symbolic evaluation

*Program slicing* is the process of taking a program $I$ and some slicing criterion $(V, n)$ (variable set $V$ and node $n$) and removing all parts of $I$ that do not affect the value, at node $n$, of any variable in $V$. Much work has focussed on the technical problems associated with slicing programs in the presence of procedures [20, 28], pointers [2, 24, 25] and jumps [3, 7, 14, 1]. This paper uses only *end slicing*, in which the end of the program is the point of interest ([23]). Thus, throughout this paper the slicing criterion is simply a set of variables.

Program slicing was initially introduced as a way of assisting debugging ([30, 26]). For this application it is important that the only simplification tool available to slicing algorithms is statement deletion. For a number of other applications, such as mutation testing ([18]) and program comprehension ([13, 15]), this restriction to statement deletion is unhelpful. In such cases *Amorphous Slicing*, which allows the application of any transformations that preserve the semantics of interest, leads to improved simplification ([13, 16, 4]).

In slicing it is possible to place a condition $C$ on the input values. Then any statement that cannot affect the values of the variables in $V$ at $n$, given that the input satisfies $C$, may be removed. This is called *conditioned slicing* ([6, 10]). In the amorphous version of conditioned slicing any transformation that preserves the effect of the original program upon the slicing criterion is valid.

Given a program $I$ and condition $C$, $S_C(I)$ shall denote the (possibly amorphous) conditioned end slice of $I$ (in which all variables are of interest) for condition $C$. Similarly, given subdomain $D' \subseteq D$, $S_{D'}(I)$ shall denote the conditioned slice in which the input is constrained to $D'$. Thus $S_{D'}(I)$ denotes $S_C(I)$ where $C(x)$ is the condition $x \in D'$.

Symbolic evaluation is the process of describing the final values of the variables in a program in terms of the initial values of the variables. Since programs normally have control-flow constructs, the result of applying symbolic evaluation to a program will usually lead to a number of symbolic values, each with a precondition.

# 4 Uniformity can help program analysis

This section describes a way in which the existence of a uniformity hypotheses may assist program analysis. This is achieved through using the information represented by the uniformity hypothesis. It thus introduces the possibility of using standard testing approaches, that generate a uniformity hypothesis, to assist program analysis.

Suppose subdomain $D'$ of $D$ has been chosen and all of the values in $D'$ are processed in the same way by $I$. Then the conditioned slice of $I$ on the subdomain $D'$ should be relatively simple. Thus, if $D^I$ were known, this would suggest an approach to program analysis: slice on the subdomains of $D^I$ and analyse these slices.

While $D^I$ is not known, it is possible to slice using the partition $D^M$, forming the set $S(I, D^M) = \{S_{D_i}(I) \mid D_i \in D^M\}$ of conditioned slices. If the uniformity hypothesis holds the slices in $S(I, D^M)$ should be relatively small. This might help solve one of the challenges of conditioned slicing: finding conditions that lead to small but useful slices.

Consider the program analysis problem of producing a proof of correctness. Then, $I$ conforms to $M$ if and only if for all $D_i \in D^M$, $I$ conforms to $M$ on $D_i$. Thus, in order to prove that $I$ conforms to $M$ it is sufficient to prove that each $S_{D_i}(I)$ conforms to $M$ on the corresponding $D_i$. It is then sufficient to consider, for each $D_i \in D^M$, $I$ and $M$ restricted to $D_i$.

The uniformity hypothesis is based on the behaviour of $M$ being relatively simple on each $D_i$. If the uniformity hypothesis holds, the $S_{D_i}(I)$ should also

be relatively simple. Thus, if the uniformity hypothesis holds, the proof of correctness has been broken down into a number of relatively simple proofs. Symbolic evaluation might be applied to each $S_{D_i}(I)$, producing an expression that can more easily be handled by an automated theorem-prover. Naturally, if the partition $D^I$ defined by $I$ were known, slicing would be applied on $D^{IM} = \{D_i \cap D_j \mid D_i \in D^M, D_j \in D^I\}$. The approach outlined in ([29]) might then be used.

Consider now an implementation $I^\Delta$ that is intended to solve the triangle problem. It thus takes three integers $x$, $y$ and $z$ and should return:

1. 'equilateral' if x=y and y=z;
2. 'isosceles' if two of x, y, and z are the same but the third is different;
3. 'scalene' if x, y, and z are all different.

The tester might analyse this specification and produce the following conditions:

$$C_1(x,y,z) \equiv x = y \wedge y = z$$

$$C_2(x,y,z) \equiv ((x = y) \vee (x = z) \vee y = z)) \wedge \neg(x = y \wedge y = z)$$

$$C_3(x,y,z) \equiv x \neq y \wedge y \neq z \wedge x \neq z$$

Suppose that the computation contained in $I^\Delta$ is the code shown below.

```
if (x==y && y==z)
   r = "equilateral";
if (x==y) r = "isosceles";
if (x==z) r = "isosceles";
if (y==z) r = "isosceles";
if (x!=y && y!=z && x!=z)
   r = "scalene";
printf("The triangle is %s \n",r);
```

Suppose that $I^\Delta$ is sliced on conditions $C_1$, $C_2$ and $C_3$. The initial step in producing a conditioned slice, of $I^\Delta$, for $C_3$ might give:

```
if (x!=y && y!=z && x!=z)
   r = "scalene";
```

This reduces to:

```
r = "scalene";
```

Similarly, the first step in the process of applying conditioned slicing with $C_2$ might give:

```
if (x==y) r = "isosceles";
if (x==z) r = "isosceles";
if (y==z) r = "isosceles";
```

This reduces to:

```
r = "isosceles";
```

Suppose conditioned slicing is applied with $C_1$. Then any effect of the first three lines is killed by the fourth line. Conditioned slicing might initially produce:

```
if (y==z) r = "isosceles";
```

Again, this may be further reduced, giving:

```
r = "isosceles";
```

The behaviour on each subdomain is quite simple. In fact, in each case it is constant. The information provided by the uniformity hypothesis has thus allowed the generation of small conditioned slices. The existence of these conditioned slices allows the production of simple proofs of correctness, for the subdomains where the behaviour is correct, and the identification of counter-examples where the behaviour is not correct. In this case it is clear that the behaviour on $C_2$ and $C_3$ is correct but that the behaviour on $C_1$ is faulty.

It is worth noting that the production of such simple slices has lent weight to the uniformity hypothesis. Thus, if producing a proof of correctness were not feasible for some subdomain, test derived using the hypothesis might be used instead.

# 5 Program analysis can help when using uniformity hypotheses

This section describes ways in which program analysis assists a tester when considering using the uniformity hypothesis. These approaches are again based upon the conditioned end slices of $I$, contained in $S(I, D^M)$, produced by slicing $I$ on the subdomains of the partition $D^M$.

If a slice $I' = S_{D_i}(I)$ is unexpectedly complex, this might indicate that $I$ takes on more than one behaviour on $D_i$. This might occur either because this subdomain should be split further or because a boundary is wrong. Then we might either further analyse this slice or test more thoroughly in $D_i$.

Let $Symb(I, D_i)$ denote the result of applying symbolic evaluation to $S_{D_i}(I)$, $D_i \in D^M$. Then $Symb(I, D_i)$ is a set of pairs, each pair $(p, f)$ consisting of a precondition $p$ and a behaviour $f$. Suppose $Symb(I, D_i)$ has been produced and it contains more than one behaviour with separate preconditions. These preconditions suggest a refinement of $D^M$: the subdomain should be partitioned into $\{\{x \in D_i \mid p(x)\} \mid \exists f.(p, f) \in Symb(I, D_i)\}$. The conditioned slices on each of these subdomains may now be produced and these should be relatively simple.

Suppose a slice $S_{D_i}(I) \in S(I, D^M)$ is simple and $Symb(I, D_i)$ contains one behaviour only. This provides some initial confidence in the behaviour of $I$ being uniform on $D_i$. It might also be possible to further analyse the relationship between the behaviour of $S_{D_i}(I)$ or $Symb(I, D_i)$ and that of $M$ on $D_i$. This analysis might, for example, involve a proof of correctness. Alternatively, it might involve determining the type of function applied. Where the form of the behaviours of $M$ and $I$ on $D_i$ is known, it may be possible to devise a test set that determines correctness on $D_i$ ([21]), thus overcoming the problem of coincidental correctness.

Consider a system designed to return the sale price of a purchase of rice and lentils. Suppose `x` denotes the amount of lentils being purchased and `y` denotes the amount of rice being purchased. The price of rice is 2 and the price of lentils is 1. There are discounts for bulk purchases: if the amount of lentils being purchases is greater than or equal to 50 there is a five percent discount and if the total price (without discount) is greater than or equal to 1000 there is a ten percent discount. The discounts are cumulative. Suppose program $I^p$, containing the following code that performs the computation, has been produced.

```
if (x >= 50.0) p1 = 0.95;
   else p1 = 1.0;
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p1*p2*(2.0*x+y);
```

There are two basic conditions, $x \geq 50$ and $y \geq 1000$, to consider. This leads to the following four conditions.

$$x < 50 \wedge (2x + y) < 1000$$

$$x < 50 \wedge (2x + y) \geq 1000$$

$$x \geq 50 \wedge (2x + y) < 1000$$

$$x \geq 50 \wedge (2x + y) \geq 1000$$

Consider the second condition, $C(x, y) \equiv x < 50 \wedge (2x + y) \geq 1000$. Then the corresponding conditioned slice of $I^p$ is

```
if (x >= 50.0) p1 = 0.95;
   else p1 = 1.0;
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p1*p2*(2.0*x+y);
```

This can be further reduced to:

```
p1 = 1.0;
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p1*p2*(2.0*x+y);
```

and then, using amorphous slicing:

```
if ((2.0*x+y) >1000.0) p2 = 0.9;
   else p2 = 1.0;
c = p2*(2.0*x+y);
```

This cannot be simplified any further. Symbolic evaluation may now be applied, leading to the following precondition/function pairs:

$$(((2x+y) > 1000.0), c = 0.9 * (2x+y))$$

$$(((2x+y) \leq 1000.0), c = (2x+y))$$

The second precondition can be simplified to $2x + y = 1000$. This analysis suggests dividing the subdomain, defined by the precondition $C(x,y) \equiv x < 50 \wedge (2x+y) \geq 1000$, into $C_1(x,y) \equiv x < 50 \wedge (2x+y) > 1000$ and $C_2(x,y) \equiv x < 50 \wedge (2a+y) = 1000$. Any test case taken from the second of these subdomains will lead to a failure.

# 6 Future Work

This paper has considered ways in which program analysis and test generation complement one another. In particular, a relationship between the uniformity hypothesis and conditioned slicing is explored. There is, however, a general principle contained in this work: information contained in test hypotheses may assist when analysing a program and program analysis may assist when using test hypotheses. This general approach may extend to other types of test hypotheses and forms of program analysis.

The potential role of program analysis, when using test hypotheses, suggests the challenge of devising test hypotheses that

1. are likely to hold;
2. lead to feasible tests that are easy to generate;
3. are relatively easy to verify using program analysis.

Other test hypotheses might represent information that can assist in program analysis. This suggests the investigation of information contained in test hypotheses, information that might assist particular forms of program analysis, and any relationships between these types of information.

# 7 Conclusions

Many test techniques make assumptions, often called test hypotheses, about the implementation under test. These allow stronger statements to be made about the effectiveness of testing if the test hypotheses hold. However, if the hypotheses do not hold then the tests generated may have little value.

Program analysis is capable of providing general information about implementations. Often, however, program complexity limits the use and effectiveness of program analysis.

This paper has considered the relationship between test hypotheses and program analysis. Within this it has concentrated on the uniformity hypothesis. Program analysis may provide confidence in or refute the test hypotheses or may suggest refinements to the hypotheses. The information provided by the existence of the uniformity hypothesis can be used to simplify program analysis through the production of small conditioned slices.

# References


[1] H. Agrawal. On slicing programs with jump statements. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 302–312, Orlando, Florida, June 20–24 1994. Proceedings in SIGPLAN Notices, 29(6), June 1994.

[2] H. Agrawal, R. A. DeMillo, and E. H. Spafford. Dynamic slicing in the presence of unconstrained pointers. In $4^{th}$ *ACM Symposium on Testing, Analysis, and Verification (TAV4)*, pages 60–73, 1991. Appears as Purdue University Technical Report SERC-TR-93-P.

[3] T. Ball and S. Horwitz. Slicing programs with arbitrary control–flow. In Peter Fritzson, editor, $1^{st}$ *Conference on Automated Algorithmic Debugging*, pages 206–222, Linköping, Sweden, 1993.

Springer. Also available as Uniersity of Wisconsin–Madison, technical report (in extended form), TR-1128, December, 1992.

[4] D. W. Binkley. Computing amorphous program slices using dependence graphs and a data-flow model. In *ACM Symposium on Applied Computing*, pages 519–525, The Menger, San Antonio, Texas, U.S.A., 1999. ACM Press, New York, NY, USA.

[5] G. V. Bochmann, A. Das, R. Dssouli, M. Dubuc, A. Ghedamsi, and G. Luo. Fault models in testing. In *Protocol Test Systems IV*, pages 17–30. Elsevier Science (North-Holland), 1992.

[6] G. Canfora, A. Cimitile, and A. De Lucia. Conditioned program slicing. In Mark Harman and Keith Gallagher, editors, *Information and Software Technology Special Issue on Program Slicing*, volume 40, pages 595–607. Elsevier Science B. V., 1998.

[7] J.–D. Choi and J. Ferrante. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems*, 16(4):1097–1113, July 1994.

[8] T. S. Chow. Testing software design modelled by finite state machines. *IEEE Transactions on Software Engineering*, 4:178–187, 1978.

[9] L. A. Clarke, J. Hassell, and D. J. Richardson. A close look at domain testing. *IEEE Transactions on Software Engineering*, 8:380–390, 1982.

[10] A. De Lucia, Anna R. Fasolino, and M. Munro. Understanding function behaviours through program slicing. In $4^{th}$ *IEEE Workshop on Program Comprehension*, pages 9–18, Berlin, Germany, March 1996. IEEE Computer Society Press, Los Alamitos, California, USA.

[11] M. C. Gaudel. Testing can be formal too. In *TAPSOFT'95*, pages 82–96. Springer-Verlag, March 1995.

[12] J. B. Goodenough and S. L. Gerhart. Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, 1:156–173, 1975.

[13] M. Harman and S. Danicic. Amorphous program slicing. In $5^{th}$ *IEEE International Workshop on Program Comprehesion (IWPC'97)*, pages 70–79, Dearborn, Michigan, USA, May 1997. IEEE Computer Society Press, Los Alamitos, California, USA.

[14] M. Harman and S. Danicic. A new algorithm for slicing unstructured programs. *Journal of Software Maintenance*, 10(6):415–441, 1998.

[15] M. Harman, C. Fox, R. M. Hierons, D. Binkley, and S. Danicic. Program simplification as a means of approximating undecidable propositions. In $7^{th}$ *IEEE International Workshop on Program Comprehesion (IWPC'99)*, pages 208–217, Pittsburgh, Pennsylvania, USA, May 1999. IEEE Computer Society Press, Los Alamitos, California, USA.

[16] M. Harman, Y. Sivagurunathan, and S. Danicic. Analysis of dynamic memory access using amorphous slicing. In *IEEE International Conference on Software Maintenance (ICSM'98)*, pages 336–345, Bethesda, Maryland, USA, November 1998. IEEE Computer Society Press, Los Alamitos, California, USA.

[17] R. M. Hierons. Testing from a Z specification. *Journal of Software Testing, Verification and Reliability*, 7:19–33, 1997.

[18] R. M. Hierons, M. Harman, and S. Danicic. Using program slicing to assist in the detection of equivalent mutants. *Journal of Software Testing, Verification and Reliability*, 9r:233–262, 1999.

[19] M. Holcombe and F. Ipate. *Correct Systems: Building a Business Process Solution*. Springer-Verlag, 1998.

[20] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–61, 1990.

[21] W. E. Howden. Algebraic program testing. *ACTA Informatica*, 10:55–56, 1978.

[22] ITU-T. *Z.500 Framework on formal methods in conformance testing*. International Telecommunications Union, 1997.

[23] A. Lakhotia. Rule–based approach to computing module cohesion. In *Proceedings of the* $15^{th}$ *Conference on Software Engineering (ICSE-15)*, pages 34–44, 1993.

[24] P. E. Livadas and A. Rosenstein. Slicing in the presence of pointer variables. Technical Report SERC-TR-74-F, Computer Science and Information Services Department, University of Florida, Gainesville, FL, June 1994.

[25] J. R. Lyle and D. Binkley. Program slicing in the presence of pointers. In *Foundations of Software Engineering*, pages 255–260, Orlando, FL, USA, November 1993.

[26] J. R. Lyle and M. Weiser. Automatic program bug location by program slicing. In $2^{nd}$ *International Conference on Computers and Applications*, pages 877–882, Peking, 1987. IEEE Computer Society Press, Los Alamitos, California, USA.

[27] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating tests. *Communications of the ACM*, 31:676–686, 1988.

[28] T. W. Reps. Solving demand versions of interprocedural analysis problems. In Peter Fritzon, editor, *Compiler Construction, 5th International Conference*, volume 786 of *Lecture Notes in Computer Science*, pages 389–403, Edinburgh, U.K., 7–9 April 1994. Springer.

[29] D. J. Richardson and L. A. Clarke. Partition analysis: A method combining testing and verification. *IEEE Transactions on Software Engineering*, 14:1477–1490, 1985.

[30] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[31] E. J. Weyuker and T. J. Ostrand. Theories of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6:236–246, 1980.

[32] L. J. White and E. I. Cohen. A domain strategy for computer program testing. *IEEE Transactions on Software Engineering*, 6:247–257, 1980.

# Annotation-Assisted Lightweight Static Checking

David Evans
evans@cs.virginia.edu
University of Virginia, Department of Computer Science

**Abstract**
While heavyweight formal methods have shown much promise in academia and remarkable success in industrial hardware projects, they are rarely used in industrial software projects. There are many reasons for this [DillRusby96, Hall96, HollowayButler96], but we believe one of the most important is the lack of a realistic adoption path between current development techniques and more formal approaches. Our research seeks to provide a few of the steps along that path. Our experience so far with LCLint [EGHT94, Evans96] indicates that lightweight static checking tools may provide an effective way to introduce formal methods into industrial environments.

## Background

There is a huge gap between the amount of effort and expertise required to use traditional development tools (such as compilers, integrated development environments, and test scripts) and formal techniques such as *Z* specifications, model checking, and program verification. On the other hand, a large class of common programming errors can be detected using simpler techniques. Our research explores what can be done with minimal programmer effort, without requiring substantial changes to traditional development processes, using a tool that requires no user interaction and runs about as fast as a typical compiler.

We have developed LCLint, a tool for statically checking C programs. LCLint provides a first step towards adoption of formal techniques and mechanical analysis. If minimal effort is invested adding annotations to programs, LCLint can perform stronger checks than can be done by any compiler or standard lint. Adding these annotations is the first step on the path to using formal analysis techniques. LCLint checking ensures that there is a clear and commensurate payoff for any effort spent adding annotations.

Some of the problems that can be detected by LCLint include: violations of information hiding; inconsistent modifications of caller-visible state; inconsistent uses of global variables; memory management errors including uses of dead storage and memory leaks; and undefined program behavior. LCLint checking is done using simple dataflow analyses. This means the checking is as fast as a compiler, and LCLint can easily be introduced into standard development cycles.

As one would expect, LCLint's performance and usability goals require certain compromises to be made regarding the checking. In particular, we believe that it is reasonable to sacrifice soundness and completeness towards these goals (see [Evans96] for a more complete argument). While this would not be acceptable in many environments, it is a desirable tradeoff in a typical industrial development environment where efficient detection of program bugs is the overriding goal. LCLint has been in active use for more than five years, and has been used by thousands of programmers in both industry and academia.

## Current Directions

Our current work focuses on extending this approach in two directions: enhancing the functionality of LCLint by adding support for user-defined annotations without relaxing the usability and efficiency requirements, and providing the next step toward heavyweight formal methods by introducing more expressive annotations and automated run-time checking.

**User-defined Annotations.** Currently, LCLint users are limited to a pre-defined set of annotations. This works well as long as their programming style is consistent with the methodology supported

by LCLint (e.g., abstract data types implemented by separate modules, pointers used in a stylized way), but is problematic if one is checking a program that does not adhere to this methodology. For example, LCLint provides annotations for checking storage that is managed using reference counting. An annotation is used to denote an integer field of a structure as the reference count, and LCLint will report inconsistencies if new pointers to the structure are created without increasing the reference count, or if the storage associated with the referenced object is not deallocated when the reference count reaches zero. If a program implements reference counting in some other way (for example, by keeping the reference counts in a separate lookup table), however, LCLint provides no relevant annotations or checking. More generally, applications often have application-specific constraints that should be checked statically. Programmers should be able to define annotations that express these decisions, and use LCLint to verify that the code is consistent with their constraints.

We are investigating extensions to LCLint that address this need by supporting user-defined annotations. Programmers will be able to invent new annotations, express syntactic constrains on their usage, and define checking associated with the annotation.

Annotations introduce state that is associated with both declarations and intermediate expressions along symbolic execution paths. The meaning of an annotation is defined by semantic rules similar to typing judgments, except they may describe more flexible constraints and transitions than is usually done with typing judgments. We are defining a general meta-annotation language that can define a class of annotations in a simple and general way. Meta-annotations define constraints and transition functions when storage is assigned, passed as parameters, returned from a function, and when control enters or exits a block or function.

We are currently experimenting with annotations for detecting buffer overflow errors. Annotations and checking needed to statically detect buffer overflows are more complex than previous LCLint annotations. They may depend on numeric constraints as well as establishing relationships between more than one reference. For example, one annotation expresses that the allocated size of storage referenced by a structure field is equal to the value of an integral field in the same structure. These annotations will provide a good basis for determining the required scope of the meta-annotation language, as well as for experimenting with the expressive requirements of the meta-annotation language.

**Towards Heavyweight Formal Techniques.** The performance and usability requirements of LCLint inherently limit the kinds of checking that can be done as well as the claims that can be made about a checked program. We can consider overcoming these limitations by relaxing these requirements. Typical users will start by using the lightweight version of LCLint, but as they become more familiar with formal techniques will be willing to invest the effort required to use heavier-weight techniques. Providing a straightforward and effective transition path from lightweight to heavyweight formal techniques is a major goal of this work.

One approach would be to require more complete specifications and use theorem-proving technology to perform more complex checking. This is similar to what is done by the Extended Static Checking (ESC) project at Compaq SRC [Detlefs98]. ESC uses a theorem proving technology, which enables it to detect a larger class of errors than can be done by the simple dataflow analyses done by LCLint, but means that the analysis is several orders slower and the size of programs that can be analyzed is severely limited. As programmers develop more complex specifications, they would be spending more time writing specifications, and checking would slow down. There would need to be several gradual transition steps between lightweight annotations with dataflow analyses, and full formal specifications with interactive theorem proving.

Another approach would be to use a combination of static and run-time checking. If LCLint is not able to prove statically that a specified property holds, it could insert run-time checks to ensure the property holds at run-time. This has the considerable disadvantage that an error may still occur at run-time, but would allow more complex properties to be guaranteed without requiring the user expertise and effort typically required of a program verification system.

Our experience using Naccio to transform programs to enforce a safety policy described using a general, high-level language [EvansTwyman99, Evans99] offers a possible approach to automatically inserting run-time checking in programs. Related approaches include the Assertion Definition Language (ADL) created by Sun Microsystems, X/Open and the Information-technology Promotion Agency (an agency of Japan's MITI) [Sankar93, Obayashi98] and Anna's Annotation Transformer [Luckham90]. Both tools generate run-time assertions from specifications. We believe a combination of run-time and static checking that builds on a lightweight base is a promising way to introduce heavyweight formal methods into industrial environments.

## Availability

More information on LCLint and source code and binary releases is available at http://lclint.cs.virginia.edu.

## Acknowledgements

LCLint grew out of work at DEC Systems Research Center and the MIT Lab for Computer Science led by John Guttag and Jim Horning. The work described in this paper is being done by David Evans, John Knight and David Larochelle at the University of Virginia.

## References

[Detlefs98] David L. Detlefs, K. Rustan M. Leino, Greg Nelson, James B. Saxe. *Extended Static Checking*. Compaq SRC Research Report #159, December, 1998.

[DillRushby96] D. Dill and J. Rushby. *Acceptance of Formal Methods: Lessons from Hardware Design.* IEEE Computer, April 1996.

[EGHT94] David Evans, John Guttag, Jim Horning and Yang Meng Tan. *LCLint: A Tool for Using Specifications to Check Code*. In *Proceedings of the SIGSOFT Symposium on the Foundations of Software Engineering*, December 1994.

[Evans96] David Evans. *Static Detection of Dynamic Memory Errors*. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (PLDI '96), Philadelphia, PA, May 1996.

[EvansTwyman99] David Evans and Andrew Twyman. *Policy-Directed Code Safety*. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May, 1999.

[Evans99] David Evans. *Policy-Directed Code Safety*. MIT PhD Thesis, October 1999.

[Hall96] Anthony Hall. *What is the Formal Methods Debate About?* IEEE Computer, 29(4):22-23, April 1996.

[HollowayButler96] C. Michael Holloway and Ricky W. Butler. *Impediments to Industrial Use of Formal Methods*. IEEE Computer, 29(4):25-26, April 1996.

[Luckham90] David Luckham. *Programming with Specifications: An Introduction to ANNA, A Language for Specifying Ada Programs*. Springer-Verlag, 1990.

[Lutz94] Robyn Lutz and Yoko Ampo. *Experience Report: Using Formal Methods For Requirements Analysis Of Critical Spacecraft Software*. In *Proceedings of the 19th Annual Software Engineering Workshop.* Greenbelt, MD, December 1994. NASA Goddard Space Flight Center.

[Nobe96] C. R. Nobe and W. E. Warner. *Lessons Learned from a Trial Application of Requirements Modeling using Statecharts.* In *Proceedings of the Second International Conference on Requirements Engineering*, pp. 86–93, April 15–18, 1996.

[Obayashi98] Masaharu Obayashi, Hiroshi Kubota, Shane P. McCarron, Lionel Mallet. *The Assertion Based Testing Tool for OOP: ADL2*. International Conference on Software Engineering, Kyoto, April 1998.

[Sankar93] Sriram Sankar and Roger Hayes. *Specifying and Testing Software Components using ADL*. Sun Microsystems, 1993.

# Analyzing Dependencies in Java Bytecode

**Jianjun Zhao**
Department of Computer Science and Engineering
Fukuoka Institute of Technology
3-10-1 Wajiro-Higashi, Higashi-ku, Fukuoka 811-02, Japan
Email:zhao@cs.fit.ac.jp

## Abstract

Understanding program dependencies in a computer program is essential for many software engineering tasks such as testing, debugging, reverse engineering, and maintenance. In this paper, we present a approach to dependence analysis of Java bytecode, and discuss some applications of our technique in Java bytecode slicing, understanding, and testing.

**Keywords**

Dependence analysis, Java virtual machine, program slicing, software testing.

## 1 Introduction

Java is a new object-oriented programming language and has achieved widespread acceptance because it emphasizes portability. In Java, programs are being compiled into a portable binary format call *bytecode*. Every class is represented by a single *class file* containing class related data and bytecode instructions*. These files are loaded dynamically into an interpreter, i.e., the Java Virtual Machine (JVM) [14] and executed. Recently, more and more Java applications are routinely transmitted over the internet as compressed class file archives (i.e., zip files and jar files). A typical example of this situation consists of downloading a web page that contains one or more applets. However, this situation leads to some problems. First, instead of class files, the source code of an application usually unavailable for the user. So when you download a program and run it, if there is some defect with it, you need to report the bug to the software developers and possibly pay for a new version of the bug-free software. However, if the developers are not available to support this software (i.e., they do not want to support the software anymore, or they are out of business), the user is the only one that can make change(s). Second, a bytecode program can have bugs since the methods used for Java software testing do not necessarily remove all possible bugs from its source program. At these cases, if we have some tools that can be used to support bytecode understanding, testing, and debugging, they can be greatly helpful for a programmer, a maintainer, and a manager.

One way to support to develop such kind of tools is program dependence analysis technique. Program dependencies are dependence relationships holding between program elements in a program that are implicitly determined by the control flows and data flows in the program. Intuitively, if the computation of a statement directly or indirectly affects the computation of another statement in a program, there might exist some program dependence between the statements. Dependence analysis is the process to determine the program's dependencies by analyzing control flows and data flows in the program.

Many compiler optimizations and program analysis and testing techniques rely on program dependence information, which is topically represented by a *dependence-based representation*, for example, a *program dependence graph* (PDG) [7, 12]. The PDG, although originally proposed for compiler optimizations, has been used for performing program slicing and for various software engineering tasks such as program debugging, testing, maintenance, and complexity measurements [2, 3, 5, 10, 16, 17]. For example, program slicing, a decomposition technique that extracts program elements related to a particular computation, is greatly benefit from a PDG on which the slicing problem can be reduced to a vertex reachability problem [16] that is much simpler than its original algorithm [18].

Dependence analysis was originally focused on procedural programs. Recently, as object-oriented software become popular, researchers have applied dependence analysis to object-oriented programs to represent various object-oriented features such as classes and objects, class inheritance, polymorphism and dynamic binding [4, 11, 13, 15], and concurrency [19, 20]. (for detailed discussions, see related work section).

However, previous work on dependence analysis has

*Throughout this paper, we use the term *bytecode program* to refer to the program generated by the compilation process; i.e., a machine-independent program written in JVM bytecode instructions, and use the term *bytecode method* to refer to the method that is written in JVM bytecode instructions and contained in a bytecode program. We also use the term *Java bytecode* to refer to bytecode programs as a whole.

mainly focused on programs written in high-level programming languages, rather than programs in low-level programming languages such as Java bytecode. Although there are several dependence analysis techniques for binary executables on different operating systems and machine architectures [6, 8, 9], the existing dependence analysis techniques can not be applied to Java bytecode straightforwardly due to the specific features of JVM. In order to perform dependence analysis on Java bytecode, we must extend existing dependence analysis techniques for adapting Java bytecode.

In this paper we propose a dependence analysis technique for Java bytecode. To this end, we first identify and define various types of primary dependencies in a bytecode program at the intraprocedural level, then we discuss some applications of our technique such as Java bytecode slicing, understanding, and testing. In addition to these applications, we believe that the dependence analysis technique presented in this paper can also be used as an underlying base to develop other software engineering tools for Java bytecode to aid debugging, reengineering, and reverse engineering.

The rest of the paper is organized as follows. Section 2 briefly introduces the Java virtual machine. Section 3 considers the dependence analysis of Java bytecode. Section 4 discusses some applications of the dependence analysis technique. Concluding remarks are given in Section 5.

## 2 The Java Virtual Machine

The Java Virtual Machine (JVM) is a stack-based virtual machine that has been designed to support the Java programming language [1]. The input of the JVM consists of platform-independent class files. Each class file is binary file that contains information about the fields and methods of one particular class, a constant pool (a kind of symbol-table), as well as the actual bytecode for each method.

Each JVM instruction consists of a one-byte **opcode** that defines a particular operation, followed by zero or more type **operands** that define the data for the operation. For example, instruction '`sipush 500`' (push constant 500 on the stack) is represented by the bytes 17,1,and 244.

For most JVM opcodes, the number of corresponding operands is fixed, whereas for the other opcodes (`lookupswitch`, `tableswitch`, and `wide`) this number can be easily determined from the bytecode context. Consequently, once the offset into a class file and the length for the bytecode of a certain method have been determined, it is straightforward to parse the bytecode instructions of this method.

At runtime, the JVM fetches an opcode and corresponding operands, executes the corresponding action, and then continues with the next instruction. At the JVM-level, operations are performed on the abstract notion of **words** [14] : words have a platform-specific size, but two words can contain values of type `long` and `double`, whereas one word can contain values of all other types. During execution of bytecode, three exceptional situations may arise:

- The JVM throws an instance of a subclass of `VirtualMachineError` in case an internal error or resource limitation prevents further execution.
- An exception is thrown *explicitly* by the instruction `athrow`.
- An exception is thrown *implicitly* by a JVM instruction.

*Example.* Figure 1 shows a simple Java class `Test` and its corresponding bytecode instructions.

## 3 Dependence Analysis

To perform dependence analysis on bytecode methods, it is necessary to identify all primary dependencies in a bytecode method. In this section, we present two types of primary intraprocedural dependencies in a bytecode method. Intraprocedural dependencies are related to dependencies in a single bytecode method.

### 3.1 Background

We give some definitions that are necessary for formally defining intraprocedural dependencies in a bytecode method from a graphical viewpoint.

**Definition 3.1** *A* digraph *is an ordered pair* $(V, A)$*, where* $V$ *is a finite set of elements called* vertices*, and* $A$ *is a finite set of elements of the Cartesian product* $V \times V$*, called* arcs*, i.e.,* $A \subseteq V \times V$ *is a binary relation on* $V$*. For any arc* $(v_1, v_2) \in A$*,* $v_1$ *is called the* initial vertex *of the arc and said to be* adjacent to $v_2$*, and* $v_2$ *is called* terminal vertex *of the arc and said to be* adjacent from $v_1$*. A* predecessor *of a vertex* $v$ *is a vertex adjacent to* $v$*, and a* successor *of* $v$ *is a vertex adjacent from* $v$*. The* in-degree *of vertex* $v$*, denoted by in-degree(*$v$*), is the number of predecessors of* $v$*, and the* out-degree *of a vertex* $v$*, denoted by out-degree(*$v$*), is the number of successors of* $v$*. A* simple digraph *is a digraph*$(V, A)$ *such that no* $(v, v) \in A$ *for any* $v \in V$*.*

**Definition 3.2** *A* path *in a digraph* $(V, A)$ *is a sequence of arcs* $(a_1, a_2, \ldots, a_k)$ *such that the terminal vertex of* $a_i$ *is the initial vertex of* $a_{i+1}$ *for* $1 \leq i \leq k-1$*, where* $a_i \in A (1 \leq i \leq k)$*, and* $k (k \geq 1)$ *is called the* length *of the path. If the initial vertex of* $a_1$ *is* $v_I$ *and*

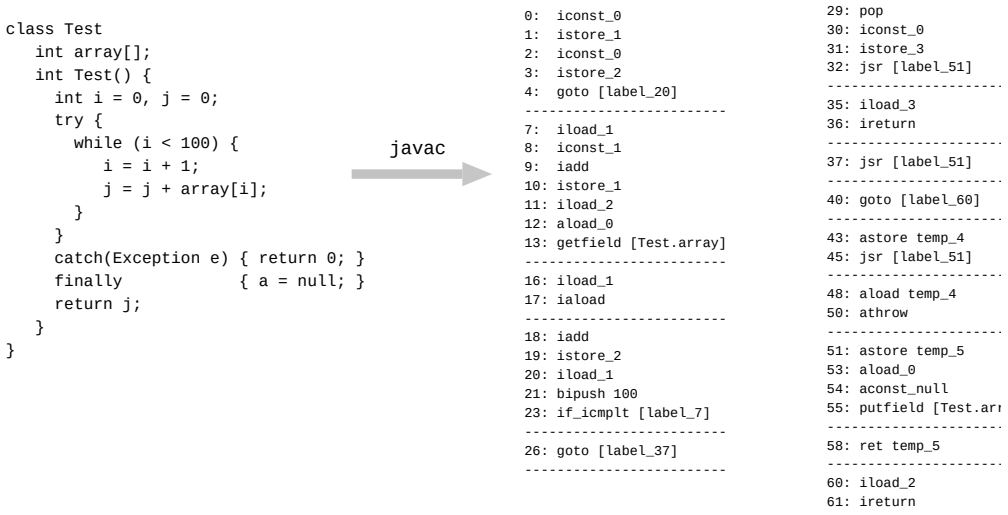



Figure 1: A simple bytecode method.

*the terminal vertex of $a_l$ is $v_T$, then the path is called a path from $v_I$ to $v_T$.*

**Definition 3.3** *A* control flow graph *(CFG) of a bytecode method $M$ is a 4-tuple $G_{cfg} = (V, A, s, T)$, where $(V, A)$ is a simple digraph such that $V$ is a set of vertices representing bytecode instructions in $M$, and $A \subseteq V \times V$ is a set of arcs which represent possible flow of control between vertices in $M$. $s \in V$ is a unique vertex, called* start vertex *which represents the entry point of $M$, such that in-degree(s) = 0, and $T \subset V$ is a set of vertices, called* termination vertices *which represent the exit points of $M$, such that for any $t \in T$ out-degree(t) = 0 and $t \neq s$, and for any $v \in V$ ($v \neq s$ and $v$ does not belong to $T$), $\exists t \in T$ such that there exists at least one path from $s$ to $v$ and at least one path from $v$ to $t$.*

Traditional control flow analysis represents each statement of a program as a vertex in its CFG. When analyzing Java bytecode, we represent a bytecode instruction as a vertex in the CFG. In our CFG, each vertex represents a bytecode instruction, and each arc represents the possible control of flow between bytecode instructions. Moreover, our CFG contains one unique vertex $s$ to represent the entry point of the method, and a set of termination vertices $T$ to represent the multiple exit points of the method. The reason for using a set of termination vertices is as follows:

In JVM, a method invocation may complete in two ways, i.e., *normal completion* if that invocation does not cause an exception to be thrown, either directly from JVM or as a result of executing an explicit `thrown` statement, and *abnormal completion* if execution of a JVM instruction within the method cause the JVM to throw an exception, and that exception is not handled within the method. Evaluation of an explicit `throw` statement also causes an exception to be thrown and, if the exception is not caught by the current method, results in abnormal method completion. Therefore, to represent these two kinds of completions of a method, our CFG uses a set of termination vertices $T$ to represent the multiple exit points of the method, that is, one for the normal method completion, and the others for abnormal method completions.

*Example.* Figure 2 shows the CFG of the bytecode instructions in Figure 1.

**Definition 3.4** *Let $u$ and $v$ be two vertices in the CFG of a bytecode method. $u$* forward dominates *$v$ iff every path from $v$ to $t \in T$ contains $u$. $u$* properly forward dominates *$v$ iff $u$ forward dominates $v$ and $u \neq v$. $u$* strongly forward dominates *$v$ iff $u$ forward dominates $v$ and there exists an integer $k$ ($k \leq 1$) such that every path from $v$ to $t \in T$ whose length is greater than or equal to $k$ contains $u$. $u$ is called the* immediate forward dominator *of $v$ iff $u$ is the first vertex that properly forward dominates $v$ in every path from $v$ to $t \in T$.*

**Definition 3.5** *A* definition-use graph *(DUG) of a bytecode method $M$ is a 4-tuple $G_{dug} = (G_{cfg}, \Sigma, D, U)$, where $G_{cfg} = (V, A, s, T)$ is a CFG of $M$, $\Sigma$ is a finite set of symbols, called* local variables *in $M$, $D : V \rightarrow P(\Sigma)$ and $U : V \rightarrow P(\Sigma)$ are two partial functions from $V$ to the power set of $\Sigma$.*

```
Building CFG...
Basic block list for: Test
--------------------------
Basic block for v0
Predecessors:
Successors:
   0:  iconst_0
   1:  istore_1
   2:  iconst_0
   3:  istore_2
   4:  goto [label_20]
-------------------------
Basic block for v1
Predecessors:
Successors:
   7:  iload_1
   8:  iconst_1
   9:  iadd
   10: istore_1
   11: iload_2
   12: aload_0
   13: getfield [Test.array]
-------------------------
Basic block for v2
Predecessors:
Successors:
   16: iload_1
   17: iaload
-------------------------
Basic block for v3
Predecessors:
Successors:
   18: iadd
   19: istore_2
   20: iload_1
   21: bipush 100
   23: if_icmplt [label_7]
-------------------------
Basic block for v4
Predecessors:
Successors:
   26: goto [label_37]
-------------------------
Basic block for v5
Predecessors:
Successors:
   29: pop
   30: iconst_0
   31: istore_3
   32: jsr [label_51]
Basic block for v6
Predecessors:
Successors:
   35: iload_3
   36: ireturn
-------------------------
Basic block for v7
Predecessors:
Successors:
   37: jsr [label_51]
-------------------------
Basic block for v8
Precedessors:
Successors:
   40: goto [label_60]
----------------------
Basic block for v9
Predecessors:
Successors:
   43: astore temp_4
   45: jsr [label_51]
----------------------
Basic block for v10
Predecessors:
Successors:
   48: aload temp_4
   50: athrow
-----------------------
Basic block for v11
Predecessors:
Successors:
   51: astore temp_5
   53: aload_0
   54: aconst_null
   55: putfield [Test.array]
-------------------------
Basic block for v12
Predecessors:
Successors:
   58: ret temp_5
-----------------------
Basic block for v13
Predecessors:
Successors:
   60: iload_2
   61: ireturn
```

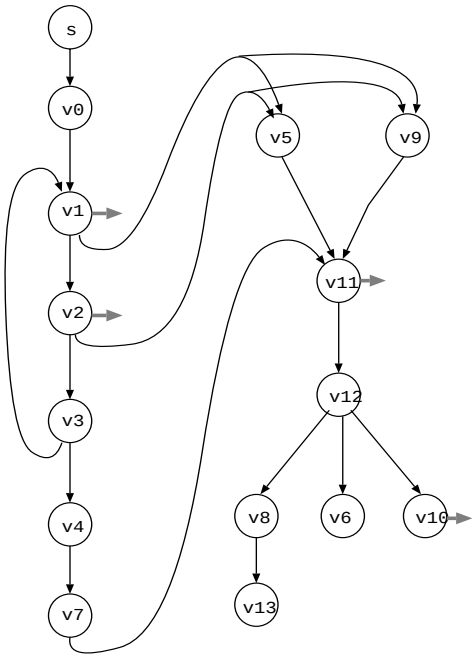



Figure 2: The CFG of the bytecode instructions in Figure 1.

The functions $D$ and $U$ map a vertex in $G_{cfg}$ to the set of local variables defined and used, respectively, in the instruction represented by the vertex. A local variable $x$ is defined in an instruction $i$ if an execution of $s$ assigns a value to $x$, while a variable $x$ is used in an instruction if an execution of $s$ requires the value of $x$ to be evaluated.

Based on the CFG and/or DUG of a bytecode method, we can define intraprocedural dependencies, i.e., control dependence and data dependence in the method.

### 3.2 Control Dependencies

#### *3.2.1 Definition of Control Dependency*

**Definition 3.6** *Let $G_{cfg} = (V, A, s, T)$ be the CFG of a bytecode method, and $u$, $v \in V$ be two vertices of $G_{cfg}$. $u$ is* directly strongly control-dependent *on $v$ iff there exists a path $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ from $v$ to $u$ such that $P$ does not contain the immediate forward dominator of $v$ and there exists no vertex $v'$ in $P$ such that the path from $v'$ to $u$ does not contain the immediate forward dominator of $v'$. $u$ is* directly weakly control-dependent *on $v$ iff $v$ has two successor $v'$ and $v''$ such that there exists a path $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ from $v$ to $u$ and any vertex $v_i$ ($1 `i \leq n$) in $P$ strongly forward dominates $v'$ but does not strongly forward dominate $v''$.*

Control dependencies represent control conditions on which the execution of an instruction depends in a bytecode method. Informally, an instruction $u$ is directly control-dependent on a control transfer instruction if whether $u$ is executed or not is directly determined by the evaluation result of $v$.

#### *3.2.2 Determining Control Dependencies*

Control dependencies represent bytecode instructions related to control conditions on which the execution of an instruction depends. There are three types of JVM instructions that may cause control dependencies.

First, JVM has control transfer instructions that can

cause conditionally or unconditionally the JVM to continue execution with an instruction other than the one following the control transfer instructions. Therefore, these kind of instructions can cause control dependencies.

- Unconditional branch instructions: `goto`, `goto_w`, `jsr`, `jsr_w`, and `ret`.
- Conditional branch instructions: `ifeq`, `iflt`, `ifle`, `ifne`, `ifgt`, `ifge`, `ifnull`, `ifnonnull`, `if_icmpeq`, `if_icmpne`, `if_icmplt`, `if_icmpgt`, `if_icmple`, `if_icmpge`, `if_acmpeq`, `if_acmpne`, `lcmp`, `fcmpl`, `fcmpg`, `dcmpl`, `dcmpg`.
- Compound conditional branch instructions: `tableswitch` and `lookupswitch`.

Second, in JVM, when the execution of a method is finished, the method must return the control to its caller. The caller is often expecting a value from the called method. JVM provides six return instructions for this purpose, which include `ireturn`, `lreturn`, `freturn`, `dreturn`, `areturn`, and `return`. Since these return instructions can also change the flow of control for the instruction execution, they form another source of control dependencies.

Third, another kind of special branch is the `jsr`, for jump subroutine. It is like a `goto` that remembers where it came from. When `jsr` is executed, it branches to the location specified by the label, and it leaves a special kind of value on the stack called a `returnAddress` to represent the return address. This may cause some control dependence.

Fourth, exceptions are sort of super-`goto` which can transfer control not only within a method, but even terminate the current method to find its destination further up the Java stack. Instructions that may explicitly or implicitly throw an exception can also cause control dependencies because it can explicitly or implicitly change the control flow from one instruction to another. These kind of instructions form another source of control dependencies.

### 3.3 Data Dependencies

#### *3.3.1 Definition of Data Dependency*

**Definition 3.7** *Let $G_{cfg} = (V, A, s, T)$ be the CFG of a bytecode method, and $u$, $v \in V$ be two vertices of $G_{cfg}$. $u$ is* directly data-dependent *on $v$ iff there exists a path $P = (v_1 = v, v_2), (v_2, v_3), \ldots, (v_{n-1}, v_n = u)$ from $v$ to $u$ such that $(D(v) \cap U(v) - D(P') \neq$ where $D(P') = D(v_2) \cup \ldots \cup D(v_{n-1})$.*

Data dependencies represent the data flow between instructions in a bytecode method. Informally an instruction $u$ is directly data-dependent on another instruction $v$ if the value of a variable computed at $v$ has a direct influence on the value of a variable computed at $u$.

#### *3.3.2 Determining Data Dependencies*

We can compute data dependencies by determining the definition and use information, i.e., the set $D$ and $U$ of each instruction first, and compute data dependencies based on such kind of information.

In order to define the data dependencies in a bytecode method, we use an annotated CFG, namely, the DUN, whose vertices are as the same as its CFG, and annotated in two functions according to the definition 2.5. First, there is a function $D(v)$ for the set of all local variables defined at vertex $v$. Second, there is a function $U(v)$ for the set of all local variables used at vertex $v$. To construct the DUG of a bytecode method, we should define these two functions explicitly. Intuitively, a use of a local variable corresponds to reading the value of that variable, whereas a definition of a local variable corresponds to writing a value into it.

According to JVM, once a method is invoked, a fixed-size frame is allocated, which consists of a fixed-sized operand stack and a set of local variables. Effectively this latter set consists of an array of words in which local variables are addressed as word offsets from the array base.

First, we can determine the definition information, i.e., the set $D$, of each instruction in a bytecode method as follow:

- A bytecode instruction that assigns a value to a local variable in this frame forms a definition of that variable. Therefore, the instructions `istore_<n>`, `istore`, `iinc`, `fstore_<n>`, `fstore`, `astore_<n>`, and `astore` form definitions of the local variable that is defined either implicitly in the opcode or explicitly in the next operand byte (or two bytes if combined with a `wide` instruction). Similarly, because the instructions `dstore_<n>`, `dstore`, `lstore_<n>`, and `lstore` effectively operate on two local variables (viz. a data item of type `long` or `double` at `n` effectively occupies local variables `n` and `n+1`), we let each such instruction form two definitions of local variables.

  For example, instruction `iinc 5 1` forms a definition of local variable 5, while `dstore_0` forms definition of both local variables 0 and 1.
- The parameter passing mechanism of bytecode causes another source of definitions of local variables: if $w$ words of parameters are passed to a particular method, then invoking that method forms initial definitions of the first $w$ local variables. The types of these parameters can be easily determined from the bytecode context. For an instance

method, the first parameter is a reference `this` to an instance of the class in which the method is defined. Types of all other arguments are defined by the corresponding method descriptor.

Second, we can determine the use information, i.e., the set $U$, of each instruction in a bytecode method as follow:

- A bytecode instruction that reads the value of a local variable forms a use of that variable. Therefore, the instructions `iload_<n>`, `iload`, `iinc`, `fload_<n>`, `fload`, `aload_<n>`, and `aload` forms uses of a single local variable defined either implicitly in the opcode or explicitly in the next operand byte (or two bytes if combined with a `wide` instruction). Similarly, instructions `dload_<n>`, `dload`, `lload_<n>`, and `lload` effectively form uses of two local variables. At implementation level, each use in a particular method may be represented by two words: the address of the using instruction and the offset of the used local variable.

Once the sets $D$ and $U$ for each instruction of a bytecode method have been determined, the DUG of the method can be constructed. Based on the DUG, it is straightforward to compute data dependencies between instructions in a method.

# 4 Applications

The dependence analysis technique presented in this paper are useful for many software engineering tasks related to Java bytecode development. Here we briefly describe three tasks: bytecode slicing, understanding, and testing.

## 4.1 Bytecode Slicing

One of our purpose for analyzing dependencies in a bytecode program is to compute static slices of the program. In this section, we informally define some notions about statically slicing of a bytecode program, and show how to compute static slices of a bytecode program based on dependence analysis.

A *static backward slicing criterion* for a bytecode program is a tuple $(s, v)$, where $s$ is an instruction in the program and $v$ is a local variable used at $s$. A *static backward slice* $SS(s, v)$ of a bytecode program on a given static slicing criterion $(s, v)$ consists of all instructions in the program that possibly affect the value of the local variable $v$ at $s$.

Similarly, we can informally define some notions of forward static slicing of a bytecode program.

A *static forward slicing criterion* for a bytecode program is a tuple $(s, v)$, where $s$ is an instruction in the program and $v$ is a local variable defined at $s$. A *static forward slice* $SS(s, v)$ of a bytecode program on a given static slicing criterion $(s, v)$ consists of all instructions in the program that possibly be affected by the value of the variable $v$ at $s$.

In addition to slicing a complete bytecode program, we can also perform slicing on a single bytecode method independently based on dependence analysis of the method. This may be helpful for locally analyzing a single method.

## 4.2 Bytecode Understanding

Sometimes it is necessary to understanding a bytecode program. For example, in the case that we can only get the class files of a Java application, but can not get the source code of the application. When we attempt to understand the behavior of a bytecode program, we often want to know which local variables in which bytecode instructions might affect a local variable of interest, and which local variables in which bytecode instructions might be affected by the execution of a variable of interest in the program. As discussed above, the backward and forward slicing of a bytecode program can satisfy the requirements. On the other hand, one of the problems in software maintenance is that of the ripple effect, i.e., whether a code change in a program will affect the behavior of other codes of the program. When we have to modify a bytecode program, it is necessary to know which local variables in which instructions will be affected by a modified variable, and which local variables in which instructions will affect a modified variable. The needs can be satisfied by backward and forward slicing the bytecode program being modified.

## 4.3 Bytecode Testing

A bytecode program can have bugs since the methods used for Java software testing do not necessarily remove all possible bugs from its source program. So it is necessary to propose some testing methods for Java software at the bytecode level. Since our dependence analysis technique analyzes both control and data dependencies which represent either control or data flow properties in a bytecode program, it is a reasonable step to define some dependence-coverage criteria, i.e., test data selection rules based on covering dependencies, for testing Java software at the bytecode level.

# 5 Concluding Remarks

In this paper we presented a dependence analysis technique to Java bytecode and discussed some applications of our technique in software engineering tasks related to Java bytecode development which include bytecode slicing, understanding, and testing. In addition to these applications, we believe that the dependence technique presented in this paper can also be used as an underly-

ing base to develop other software engineering tools to aid debugging, reengineering, and reverse engineering for Java bytecode. In order to make our technique more useful, we are now extending our analysis technique to handle interprocedural dependence analysis as well as exceptions and concurrency in Java bytecode.

Now we are developing a dependence analysis tool to automatically analyze various types of primary dependencies in a bytecode program and construct the dependence graph for the program. We also intend to use the graph as an underlying representation to develop slicer and testing tool for Java bytecode.

## REFERENCES


[1] K. Arnold and J. Gosling, "The Java Programming Language," Addison-Wesley, 1996.

[2] H. Agrawal, R. Demillo, and E. Spafford, "Debugging with Dynamic Slicing and Backtracking," *Software-Practice and Experience*, Vol.23, No.6, pp.589-616, 1993.

[3] S. Bates, S. Horwitz, "Incremental Program Testing Using Program Dependence Graphs," *Conference Record of the 20th Annual ACM SIGPLAN-SIGACT Symposium of Principles of Programming Languages*, pp.384-396, Charleston, South California, ACM Press, 1993.

[4] J. L. Chen, F. J. Wang, and Y. L. Chen, "Slicing Object-Oriented Programs," *Proceedings of the APSEC'97*, pp.395-404, Hongkong, China, December 1997.

[5] J. Cheng, "Process Dependence Net of Distributed Programs and Its Applications in Development of Distributed Systems," *Proceedings of the IEEE-CS 17th Annual COMPSAC*, pp.231-240, U.S.A., 1993.

[6] C. Cifuentes and A. Fraboulet, "Intraprocedural Static Slicing of Binary Executables," *Proc. International Conference on Software Maintenance*, pp.188-195, October 1997.

[7] J.Ferrante, K.J.Ottenstein, J.D.Warren, "The Program Dependence Graph and Its Use in Optimization," *ACM Transaction on Programming Language and System*, Vol.9, No.3, pp.319-349, 1987.

[8] J. Hatcliff, J. C. Corbett, M. B. Dwyer, S. Sokolowski, and H. Zheng, "Formal Study of Slicing for Multi-threaded Programs with JVM Concurrency Primitives," *Proc. the Static Analysis Symposium*, September 1999.

[9] J. R. Larus and E. Schnarr, "EEL: Machine-independent Executable Editing," *SIGPLAN Conference on Programming Languages, Design and Implementation*, pp.291-300, June 1995.

[10] B. Korel, "Program Dependence Graph in Static Program Testing," *Information Processing Letters*, Vol.24, pp.103-108, 1987.

[11] A. Krishnaswamy, "*Program Slicing: An Application of Object-Oriented Program Dependency Graphs*," Technical Report TR94-108, Department of Computer Science, Clemson University, 1994.

[12] D. Kuck, R.Kuhn, B. Leasure, D. Padua, and M. Wolfe, "Dependence Graphs and Compiler and Optimizations," *Conference Record of the 8th Annual ACM Symposium on Principles of Programming Languages*, pp.207-208, 1981.

[13] L. D. Larsen and M. J. Harrold, "*Slicing Object-Oriented Software*," *Proceeding of the 18th International Conference on Software Engineering*, German, March, 1996.

[14] T. Lindholm and F. Yellin, "The Java Virtual Machine Specification," Addison-Wesley, 1997.

[15] B. A. Malloy and J. D. McGregor, A. Krishnaswamy, and M. Medikonda, "An Extensiblesible Program Representation for Object-Oriented Software," *ACM Sigplan Notices*, Vol.29, No.12, pp.38-47, 1994.

[16] K. J. Ottenstein and L. M. Ottenstein, "The Program Dependence Graph in a software Development Environment," *ACM Software Engineering Notes*, Vol.9, No.3, pp.177-184, 1984.

[17] A. Podgurski and L. A. Clarke, "A Formal Model of Program Dependences and Its Implications for Software Testing, Debugging, and Maintenance," *IEEE Transaction on Software Engineering*, Vol.16, No.9, pp.965-979, 1990.

[18] M. Weiser, "Program Slicing," *IEEE Transaction on Software Engineering*, Vol.10, No.4, pp.352-357, 1984.

[19] J. Zhao, J. Cheng, and K. Ushijima, "Static Slicing of Concurrent Object-Oriented Programs," *Proceedings of the 20th IEEE Annual International Computer Software and Applications Conference*, pp.312-320, August 1996, IEEE Computer Society Press.

[20] J. Zhao, "Slicing Concurrent Java Programs," *Proc. Seventh IEEE International Workshop on Program Comprehension*, pp.126-133, Pittsburgh, PA USA, May 1999.

# Testing 2

# Third Eye — Specification-Based Analysis of Software Execution Traces

**Raimondas Lencevicius** **Alexander Ran** **Rahav Yairi**
Nokia Research Center
5 Wayside Road
Burlington, MA 01803, USA
Raimondas.Lencevicius@nokia.com Alexander.Ran@nokia.com Rahav.Yairi@nokia.com

**ABSTRACT**
Testing of complex software systems that operate on platforms with limited resources and have real-time constraints is a difficult task. Third Eye is a framework for tracing and validating software systems using application domain events. We use formal descriptions of the constraints between events to identify violations in execution traces. Third Eye is a flexible and modular framework that can be used in different products. We use Third Eye for testing an implementation of the Wireless Application Protocol (WAP). Our tool is a helpful addition to software development infrastructure.

**Keywords**
Software execution tracing, system testing, specification-based testing, application logic testing, event-based specifications.

## 1 INTRODUCTION

Currently many software-intensive systems such as personal communication devices or communication network elements integrate many dozens of software components that are designed to run on different types of hardware, to interoperate with different environments and to be configurable for different modes of operation and styles of use. To complicate the situation further, these components are often developed by geographically distributed teams, using different programming languages, development tools, and even different design and development methodologies. All this makes complete testing of these systems in a lab very hard. In these circumstances, understanding what interaction between multiple software components caused a fault is an extremely tedious process. There is a definite need to update our approaches to testing such systems.

Complexity of modern software led many organizations to focus on software architecture to simplify software life-cycle management. Testing is not commonly done against architectural descriptions because a significant conceptual gap exists between typical architectural description of complex software and its implementation.

In the Third Eye project, we have defined a methodology for tracing software execution by reporting events meaningful in the application domain or essential from the implementation point of view. Many of the ideas incorporated in the Third Eye framework were inspired by the Logic Assurance system [2] and work on enforcing architectural constraints [1]. In Third Eye, we have used different technologies to make the framework more extensible, to allow its integration with other trace analysis tools and specification languages. The implemented prototype of the Third Eye framework includes reusable software components for event definition and reporting and stand-alone tools for storage and query of event traces, constraint specification and trace analysis. We also made our framework portable to a number of execution platforms.

## 2 THIRD EYE ARCHITECTURE

Tracing execution of complex software is a standard practice in many projects. However, most projects perform the tracing in an ad-hoc manner with little or no method and tool support. Often tracing is added in response to difficulties in system integration testing. In such situations, traces have neither coherent content nor format necessary for structured information storage or automatic analysis. Third Eye transforms a useful ad-hoc technique into a disciplined engineering practice.

### 2.1 Third Eye Conceptual Architecture

A central decision of the Third Eye framework is what information from the execution state of the program is traced. We decided to trace occurrences of *events*. Unlike program variables, function calls and other implementation domain constructs, events cross the boundary between application and implementation domains allowing abstract specifications that use their properties and a simple representation in the implementation domain. Such representation helps to produce traces without introducing new errors. “Event” in this case is a qualitative change in the state of an entity either meaningful in the application domain or significant architecturally.

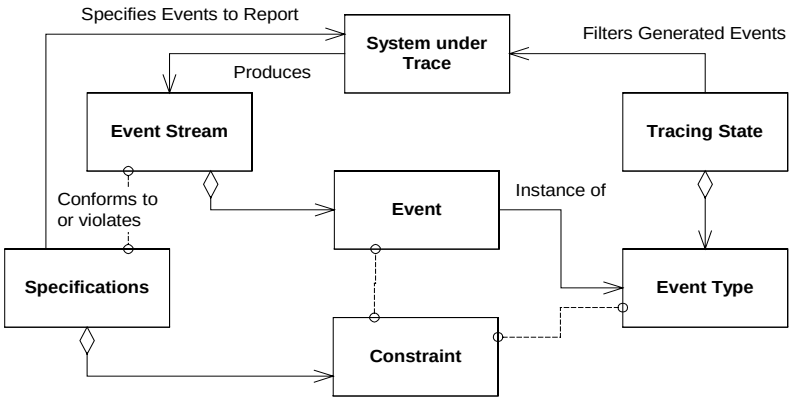



**Figure 1. Third Eye Conceptual Architecture**

Figure 1 illustrates the Third Eye conceptual architecture. In the Third Eye framework, events are typed objects. One way to implement event types is to take advantage of the programming language type system. This would support the type safety and inheritance of event manipulating code. However, we have chosen to make the event type system in Third Eye external to the programming language. This allows dynamic definition of new event types as well as sharing of event types between the event reporting and event monitoring subsystems that might reside on different platforms.

An event type has a name, a list of named and typed properties, and a type constructor. Third Eye event types are similar to classes in programming languages although the only method associated with the event type is its constructor. We allow event type inheritance. Type constructors minimize the code needed for creating new events. To report an event, developers specify the type of the event and sets values of the event properties. Developers need to set only the properties that were not set already by the event constructor.

Events in Third Eye are characterized by the time and location of their occurrence. By "location" we mean the symbolic location within the executing software. Time may be measured locally on a specific processor, but needs to have a globally meaningful interpretation in a multiprocessor system. Time/Location stamping of the events is implicit in Third Eye and is done for all events that are subtypes of `TimedEvent` and `LocalizedEvent` types. Such events have predefined properties `timestamp` and `location` that are set in a constructor.

Correct behavior specifications define constraints on the properties of the events, their sequence, location, and timing. We use formal descriptions of the constraints between events to identify violations in execution traces.

Another important concept of Third Eye is the *tracing state*. A tracing state is a set of event types generated in that state. Other event types are filtered out and not reported. The system is always in a specific tracing state. Tracing states have two important purposes. First, tracing states correspond to specifications. A program specification describes a set of constraints on events. The event types used in a specification have to be monitored to validate a trace against this specification. All event types contained in a specification and monitored for this specification form a tracing state. Consequently, there is a mapping between specifications and tracing states. This mapping allows to filter out events irrelevant to the specification.

Tracing states also control the overhead of tracing on the executing system. A control interface is provided to dynamically define and change tracing states. For example, the level of tracing detail can be increased in response to observation of an anomaly in system behavior.

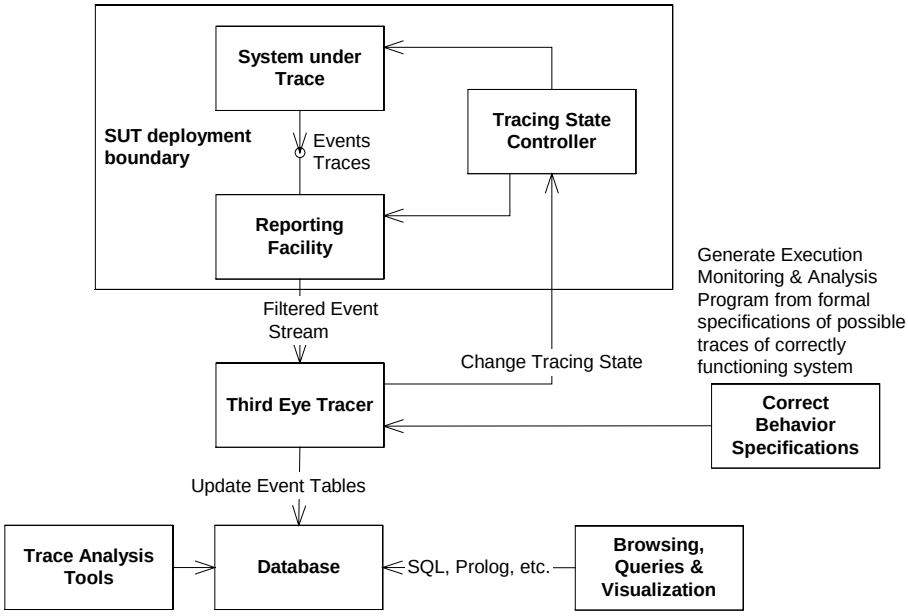



**Figure 2. Third Eye Module Structure**

### 2.2 Third Eye Modules and Interfaces

The Third Eye framework includes modules for event type definition, event generation and reporting, tracing state definition and management, specification-based trace monitor generation, trace logging, query and browsing interfaces and trace visualization (Figure 2). Modules of event type definition, event reporting facility and tracing state controler are integrated with the software of the system under trace (SUT). The rest of the modules are independent from the SUT and can be deployed on a different execution platform to minimize the influence on system performance.

The Third Eye module structure is designed for different modes of operation. Users do not need to learn and manage modules of the framework that are not relevant to their project. Thus, for example, it is possible to use the Third Eye framework only to store a stream of events in relational database without correct behavior specification and analysis components.

The module structure was partitioned along the lines of standard interfaces to achieve portability and to enable integration to third party software and tools. For example, the event reporter and tracer can be connected through a file, a socket, or using an ORB. Trace delivery for logging and analysis uses alternative interfaces to accommodate devices with different data storage and connectivity capabilities.

## 3 USING THIRD EYE

We have implemented a Third Eye framework prototype that is currently used by the Third Eye project team in collaboration with product development teams in Nokia's business units. Figure 3 illustrates Third Eye's use in the software development process.

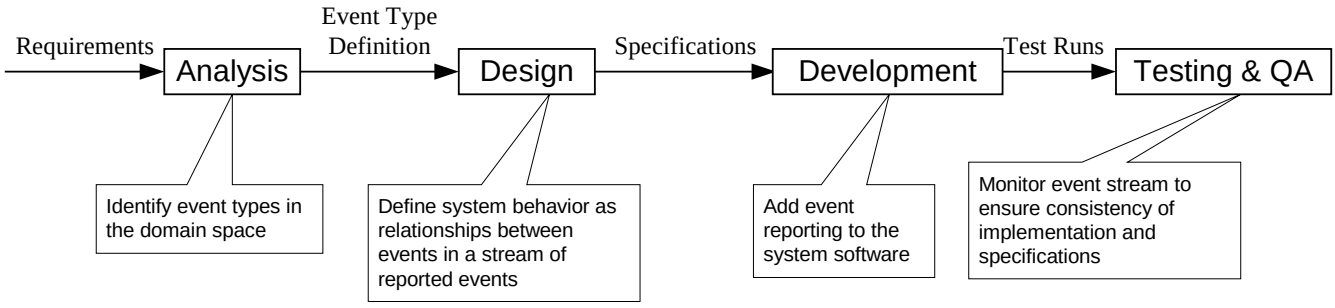



**Figure 3. Third Eye in Software Development Process**

We used Third Eye to test a number of software systems: the memory subsystem of one of Nokia's handsets, Apache Web Server, and WAP (Wireless Application Protocol) [3] client. In this section, we describe the testing of WAP client protocol layers and conformance to the *logical scopes* design discipline explained below.

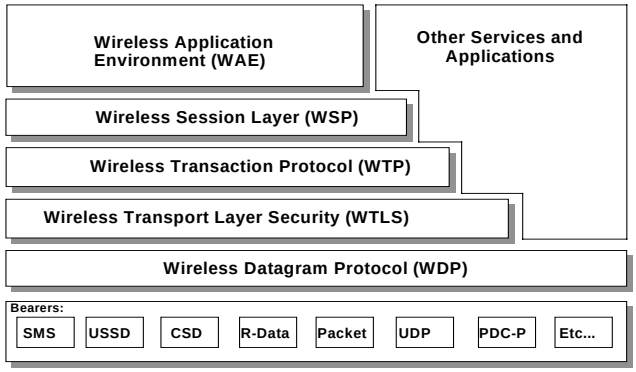



**Figure 5. WAP layered architecture**

The Wireless Application Protocol (WAP) is an industrial standard for applications and services that operate over wireless communication networks. WAP layered architecture (Figure 5) provides an application layer through Wireless Session Protocol (WSP) that interfaces to session services. A connection-oriented service operates above the Wireless Transaction Protocol (WTP) layer. The WTP runs on top of a datagram service.

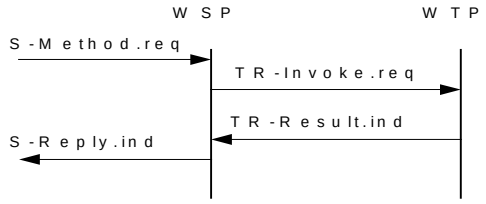



**Figure 6. WSP-WTP primitive sequence for request-response**

### 3.1 Validating WAP primitive sequences

When an application requests information through the WAP protocol, such request passes through protocol layers as a sequence of primitives defined by the WAP standard. Specifically, a request for information by an application is handled as a method invocation at the session layer (WSP). In turn, this method invocation is translated into a transaction at the transaction layer (WTP) (Figure 6). In detail, the invocation of an S-Method request is followed by the invocation of the TR-Invoke request. When the transaction returns a TR-Result result, it is used in S-Reply that forwards received information to the application. We have simplified the message diagram in Figure 6 for clarity purposes. The full message sequence contains additional confirmations and acknowledgements.

Although the message sequence in the protocol specification is rather simple, it spans two protocol layers, a number of implementation files and functions, and is not easy to validate. Furthermore, the protocol allows many outstanding method invocations and many outstanding transactions, making user tracking of the protocol state very difficult. Third Eye simplifies this process. With Third Eye users add events in the functions that correspond to the protocol primitives and then check whether the event sequence corresponds to the protocol message sequence. To monitor the WSP-WTP primitive sequence, we defined the following event types:

```
te_event_type("S-Method.req",
   TIMED, "Transaction ID", INTEGER)
te_event_type("TR-Invoke.req",
   TIMED, "Transaction ID", INTEGER)
te_event_type("TR-Result.ind",
   TIMED, "Transaction ID", INTEGER)
te_event_type("S-Reply.ind",
   TIMED, "Transaction ID", INTEGER)
```

The first parameter of definitions specifies the type name of the event, for example "`S-Reply.ind`". The second parameter indicates the constructor to be used, in this case the `TIMED` constructor that sets an inherited timestamp property. The third and fourth parameters specify the first property of the event, which for all types above is called `Transaction ID` and is an integer.

After defining event types, the event invocations are placed in corresponding functions and events are reported during testing in an event trace. The event stream can be read from a socket or from a storage device like a file. Events are mapped to SQL statements and stored in a database. Events can also be mapped to Prolog clauses to be stored in a Prolog fact file. These mappings are simple because Third Eye event structure corresponds to SQL tables and Prolog predicates. The tracer has a graphical user interface that allows the system tester to change the tracing state during runtime.

Events and event type information are stored in an SQL database through the ODBC interface. System testers can use DBMS query and reporting tools to visualize system behavior and to browse the stored event information. This provides an interface to find errors without writing a system specification.

Third Eye checks the trace using constraints that specify the correct event sequence:

```
method_with_result(TransactionID) :-
  S-Method.req(TransactionID, Time1),
  TR-Invoke.req(TransactionID, Time2),
  TR-Result.ind(TransactionID, Time3),
  S-Reply.ind(TransactionID, Time4),
  Time1 < Time2 < Time3 < Time4.
```

This constraint is expressed as a Prolog rule. The constraint requires the protocol primitives to follow each other in a correct order. When Third Eye checks a trace, it finds all events corresponding to the constraint and alerts the user if any events do not satisfy the constraint. In this example, the trace will contain events of four types defined above:

```
S-Method.req(1, 100)
TR-Result.ind(2, 150) // Violation
TR-Invoke.req(1, 200)
TR-Result.ind(1, 300)
S-Reply.ind(1, 400)
```

Third Eye will match all events corresponding to the given constraints. In our example, it will match four events with transaction number 1. These events are tagged as conforming to the specification. The events that remain untagged after matching violate the constraints. In this example, `TR-Result.ind(2, 150)` event violates the constraint, and the system informs the user that program behavior contains errors. Since this method of constraint testing would flag all events of types that do not appear in a constraint, such events should be excluded from the trace by using a tracing state.

### 3.2 Logical scopes in WAP implementation

While investigating the WAP implementation as well as implementations of other software systems, we became aware of the general design problem of resource management. Resources, especially in embedded systems, are limited. Resources that are allocated and not released or that are released too late could cause a decrease of performance or a system crash. The objective is to design a methodology in which resource management is explicit, easy to maintain and where violations can be easily identified.

In many cases, system functionality can be characterized as a set of nested tasks that have beginning and end. To perform a task, a system requires certain resources such as memory, processor time, input focus, persistent data locks, or virtual paths. Functional tasks establish scopes in which logical and physical resources are allocated and freed. The scope can be establishing and terminating a call on a mobile phone, handling an authentication request on a security server, establishing and terminating a virtual connection on an ATM node and so on. There is usually no single control scope such as a function call and return that delimits the life span of the task. This makes it hard to determine which resources were allocated to perform this specific task, when the task has been completed, and when resources can be freed.

The key to our approach to managing resources is the notion of a *logical scope*. A logical scope is the time span between beginning and termination of a task. Although it may be hard to represent logical scopes in the structure of the software, it is usually possible to identify where a logical scope begins and where it terminates. It is then possible to define events that represent beginning and termination of different logical scopes. One can also create specifications of resources that are required within a scope. If allocation and release of these resources is reported to Third Eye, the Third Eye framework can monitor correctness and efficiency of resource management within logical scopes. A generic rule for logical scope validation is *validate_scope*:

```
validate_scope :-
  task_begin(TaskID, TaskBeginTime),
  resource_allocation(ResourceID,
        TaskID, AllocTime),
  resource_free(ResourseID, FreeTime),
  task_end(TaskID, TaskEndTime),
  TaskBeginTime < AllocTime < FreeTime
    < TaskEndTime.
```

In the WAP implementation, there are numerous places where the logical scope model is used. For example, in the WTP layer a transaction handle is allocated at the beginning of a transaction and deallocated at the end. Packets sent in a transaction by the WTP adaptation layer are allocated before sending and deallocated when the next message is sent (Figure 7). Packet allocation, transmission, and deallocation form a nested scope inside the transaction scope. Neither the transaction scope, nor the packet sending subscope corresponds to a single textual entity in the program. Both encompass a number of function calls, internal message sends, and are intertwined with other logical scopes and layers. Consequently, enforcing the rules of logical scope design, i.e. deallocation of resources allocated in the beginning of a scope, is difficult using conventional means. However, Third Eye allows a simple validation of the design by inserting allocation and deallocation events together with the logical scope boundary events and specifying design constraints. The rules mirror similar rules without logical scopes. For example, a general packet deallocation rule is:

```
packet_deallocate_spec
 (Packet, AllocTime, DeallocTime) :-
  packet_allocate(Packet, AllocTime),
 packet_deallocate(Packet, DeallocTime),
  DeallocTime > AllocTime.
```

Adding the logical scope boundaries simply adds a condition:

```
packet_deallocate_logical(Packet) :-
  packet_deallocate_spec
   (Packet, AllocTime, DeallocTime),
  same_scope_follows(AllocTime,
    DeallocTime).

same_scope_follows(Time1, Time2) :-
  scope_begin(ScopeBeginTime),
  scope_end(ScopeEndTime),
ScopeBeginTime<Time1<Time2<ScopeEndTime.
```

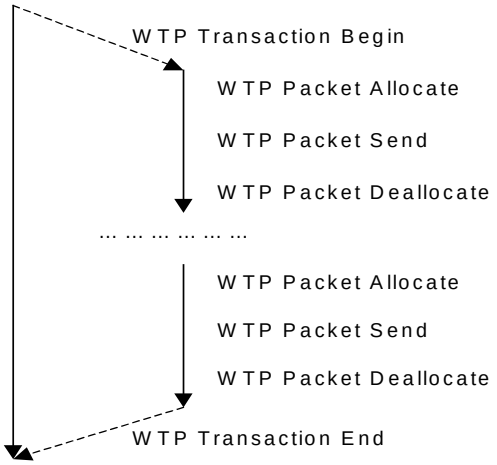



**Figure 7. WTP transaction scope and nested packet allocation sub-scope**

### 3.3 Global constraints in WAP

Although previous sections dealt with properties of the protocol expressed as sequences of events, Third Eye can monitor much wider variety of constraints. For example, the tool can check a property that all packets with low sequence numbers arrive within a certain amount of time:

```
lost_packet(PacketNumber) :- not(
  packet_received(PacketNumber, T)),
  PacketNumber < LOW_BOUNDARY.
```

The Third Eye can also check whether the number of maximum outstanding method requests in a session layer is less than the maximum possible value. Violation of such constraints could be difficult to spot with conventional tools.

## 4 CONCLUSIONS

Third Eye can be used for debugging, monitoring, specification validation, and performance measurements. All these different scenarios use typed events—a concept simple and yet expressive enough to be shared by product designers and developers. Users can achieve the system-testing goals without delving into complicated concepts like formal methods or database programming.

The Third Eye is designed with an open architecture. Therefore, the third party tools, including databases, analysis and validation tools, can be easily added or exchanged.

Several features of our prototype framework were essential to make it practical to use in product development:

- **Portability—**by defining a simple API for the SUT and by not relying on a specific interface between the SUT and the Tracer, Third Eye can be integrated into many different systems throughout Nokia. The trace structure conforms to the data-management tool formats and allows automatic trace analysis for specification verification.
- **Low overhead—**event processing is mostly done outside the SUT. Adding a dynamic filtering allows the user to control which events will be reported back to the tracer.

We believe that the Third Eye is a practical framework for specification-based analysis and adaptive execution tracing of software systems.

REFERENCES


[1] R. Balzer, Enforcing Architecture Constraints, SIGSOFT'96 Workshop, San Francisco, CA, pp. 80-82, 1996.

[2] Shtrichman, O.; Goldring, R., The 'Logic Assurance (LA)' system-a tool for testing and controlling real-time systems. *Proceedings of the Eighth Israeli Conference on Computer Systems and Software Engineering*, pp. 47–55, 1997.

[3] WAP protocol, WAP forum, http://www.wapforum.com, 2000.

# Testing, Proof and Automation. An Integrated Approach

Simon Burton John Clark John McDermid
Department of Computer Science.
University of York,
Heslington, York.
YO10 5DD, England.
+44 1904 432749
{burton, jac, jam}@cs.york.ac.uk

## Abstract

This paper presents a discussion on the complementary roles of testing and proof within automated software verification and validation processes. We demonstrate how a combination of the two approaches can lead to greater levels of automation and integrity. In particular we discuss the use of automated counter-example generation to support proof, and automated proof as a means of automating and checking test case generation. The high levels of automation are made possible by identifying repeating structures in the proofs, restricting the specification to a subset of an otherwise expressive formal notation and exploiting a general–purpose theorem proving tool with built-in constraint solvers.

## 1 Introduction

In the past, testing and proof have not been easy bedfellows. Despite their shared goal of increased software quality, proof has been seen as being for the cognoscenti, testing for software engineering's working class. The authors believe that this artificial dichotomy is harmful and that testing and proof can be used together to good effect. Even without the benefits of formal refinement, formal specifications can contribute greatly to the quality of a software product. They allow for a concise, unambiguous and explicit specification of the desired behaviour of the system. As such, they are a good basis for automated test activities. Additionally, testing to generate counter-examples to proofs can save much effort and produce illustrative examples for debugging.

The use of formal specifications themselves is still seen by many as a barrier to the widespread industrial usage of formal methods. For the benefits of formal methods in V&V to be fully exploited in industry there is a need to "disguise" the formality in some way [15]. Recent work [5] has shown that formal specifications and the corresponding proof obligations for specification validation can be generated from more intuitive engineering notations with mathematical underpinnings. Such an approach not only enables engineers with the domain knowledge to use specification notations they are comfortable with, but the translation to formal specification has the effect of restricting the subset of the formal notation used and imposes regular structures on the proofs that need to be discharged to validate certain properties (such as completeness and determinism) in the specification. These restrictions, coupled with the subset of data types used for particular domains can be exploited to develop powerful targeted heuristics for automating the V&V activities. The approaches discussed in this paper are assumed to be undertaken in the context of formal specifications generated in this manner.

In the rest of the paper we describe how a combination

of testing, proof and restricted structures in the specification can be used to enhance both the integrity and automation of several areas of the software verification and validation process. This symbiotic relationship between testing and proof is made feasible by extending previous work on testing from formal specifications and making use of a flexible theorem proving tool with integrated constraint solvers.

The paper is structured as follows. Section 2 discusses the role of testing in the automatic generation of counter-examples to proofs. Section 3 describes how proof can be used as a means of verifying automated test case generation strategies and also as a means of performing the automation itself. We also show how more effective testing strategies can be developed based on the automatic generation of formally specified test cases and how proof can be used as a testing oracle. Section 4 summarises the main contributions of the work and presents some conclusions.

## 2 Testing and Proof

Proof conjectures can arise at various points in the V&V process. For example, to ensure that a specification satisfies certain "healthiness" criteria such as completeness and determinism or to verify that a program is a correct refinement of its formal specification. In all cases, invalid conjectures can waste a large amount of proof effort. Therefore, before a long and arduous manual proof is embarked upon it is re-assuring to have a good degree of confidence in the validity of the conjecture. Use of constraint solving techniques to generate counter-examples not only saves proof effort but can provide illustrative information to use when tracking the fault. The generation of counter-examples to verify properties of a specification couched in terms of proofs is a form of testing. Typically sample data are generated and then tested to see whether they break the specification. If this is the case, a counter-example has been found.

Constraint solving in general is known to be intractable [13]. However, in practical situations, one never needs to solve "general" constraints but a particular subset that have restricted structures and particular input space characteristics. These properties can be exploited to automate the search for counter-examples. The authors use the Z [16] type checker and theorem prover CADiℤ [19] to automate this task. In this sense our usage of CADiℤ is similar to that of the Nitpick Z-based specification checker [12] that used model-checking techniques to generate counter-examples to specification assertions. However, CADiℤ has the additional flexibility that general purpose proof tactics can be written (using a lazy functional notation [20]), that can be invoked interactively from within the tool and applied to any proof obligation on the screen. Proof tactics have been written that attempt a best effort at automatically proving conjectures of certain types (e.g. completeness checks). If the proof fails or is inconclusive, the tactics then perform some simplification to transform the conjecture into a suitable form for the integrated constraint solvers. A number of constraint solvers can then be invoked to attempt counter-example generation, these include a model-checker (SMV [3]) and a simulated annealing based heuristic search [6]. The amount of simplification required before the constraint solvers can be efficiently applied will depend on the structure of the proof obligations.

Such automated proof tactics have been used to good effect when a large number of similar proof conjectures were needed to be solved [5]. A situation which would have otherwise been time consuming if done manually and could have led to "reviewer blindness" leading to missed error cases. Different constraint solvers have been found to be effective for different input domains. For example model-checking is only practical for discrete input domains, whereas optimisation-based search techniques are also suited to infinite state spaces and non-linear constraints.

## 3 Proof and Testing

Formal specifications are a good basis for testing. They allow for a concise and unambiguous representation of the requirements and are amenable to proof and automated analysis. Test generation techniques for model-based formal specifications [14, 8, 17] such as Z [16] or VDM-SL [10] are typically based on the principle of partitioning the specification into equivalence classes [9]. Equivalence classes are partitions of the specification input space that are assumed, for the purpose of testing, to represent the same behaviour in the specification. Such techniques are

amenable to automation and tool support. However, as in all cases where automation is introduced, and especially for high integrity systems, the integrity of such tools is of great importance. For automated testing to be able to provide confidence in the conformance of the software to its specification, the test generation strategies must be both verified and validated. In other words, they must not only be shown to be correctly implemented but must also be shown to be adept at finding errors in the implementation.

### 3.1 Verification of Automated Testing Strategies

There are various criteria that can be used when verifying that test partitioning strategies have been correctly implemented. For example, the tests can be shown to completely cover the valid input space of the original specification. If this were not the case, important parts of the implementation, that could possibly contain faults might remain untested. If the resulting tests are represented using the same formal notation as the original specification, these verification activities can be performed using proof. The completeness of the generated tests ($T_1 .. T_n$) with respect to the original specification ($Spec$) can be verified by proving a conjecture of the following form:

$$Theorem1 :$$
$$\vdash \forall Inputs \bullet Spec \Leftrightarrow T_1 \lor ... \lor T_n$$

An example partitioning strategy identifies expressions in the specification of the form $A \lor B$ and partitions these into the following test cases $A \land B$, $\neg A \land B$ and $A \land \neg B$ [8] where $A$ and $B$ could be complex predicates themselves. The conjecture used to prove that these partitions preserve the valid input state-space of the original specification would therefore take the following form:

$$Theorem2 :$$
$$\vdash \forall Inputs \bullet A \lor B \Leftrightarrow$$
$$(A \land B) \lor (\neg A \land B) \lor (A \land \neg B)$$

This theorem can be proven in a few simple steps. The same proof steps can be used regardless of the structure of the expressions represented by $A$ and $B$. In general, a proof can be derived for each partitioning strategy and used to verify the outcome each time that strategy is applied. Using the proof tactic mechanism in CADiℤ the authors have automated these proofs for a number of common partitioning strategies. Whenever a strategy is applied, the corresponding correctness proof can be automatically invoked on the result. This ensures that, whatever the means of test generation, the *result* can always be shown to be valid or otherwise. The tool can be instructed to record the individual proof steps taken in applying a proof tactic and these can be printed in a form amenable to human scrutiny. Therefore, if the tool cannot be trusted, a rigorous argument can be developed to support the validity of the proof steps.

Given a formal definition of a testing strategy as an equivalence (e.g. *Theorem 2* above), the derivation of the test cases themselves can also be automated using general purpose proof tactics. The principle is similar in operation to the use of Disjunctive Normal Form (DNF) to simplify an expression into a disjunction of conjuncts that can each be used as separate test cases. Where conversion to DNF uses simple logic rewrite rules to distribute disjunctions, more targeted equivalences can be formulated based on common testing heuristics.

Test partitioning based on the formal specification of the testing heuristics has been implemented using CADiℤ proof tactics. Generic partitioning strategies are specified as equivalences in the form of *Theorem 1*. A proof tactic is invoked upon the predicate to be partitioned to instantiate the generic equivalence with the operands of the predicate and simplify the whole specification to reveal a disjunction of partitions. Each test case is equivalent to the original specification where the input space has been constrained according to one of the partitions. The completeness of the partitioning strategy is left as a side conjecture to prove to ensure that the partitioning was valid. This can be automated by extending the partitioning tactic with the general purpose proof for the strategy as described above. The following simple example demonstrates how test partitions are derived. The example specification calculates the square root ($r!$) of a positive integer ($n?$) and the test partitions are generated using boundary value analysis of the $\geq$ operator (based on the premise that errors often occur on or around the boundary [2]).

The specification is given as the following Z schema[1]

[1] ? and ! are Z convention for inputs and outputs respectively.

*SquareRoot*

$n?, r! : \mathbb{R}$

---

$n? \geq 0 \land$
$r! \times r! = n?$

The boundary value analysis test heuristic for real numbers is specified as the following equivalence. Note that the "just-off" the boundary case is chosen here as 0.1. This value may vary for different applications and would chosen by the tester based on various application attributes such as the resolution of the concrete types used to implement the abstract $\mathbb{R}$ type.

$$\vdash \forall x, y : \mathbb{R} \bullet x \geq y \Leftrightarrow$$
$$(x = y) \lor (y < x \leq y + 0.1) \lor (x > y + 0.1)$$

An un-partitioned test specification for the schema is described as an existential quantifier as follows:[2]

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n?$$

The partitioning theorem is now introduced and instantiated with the local operands. The partitioning theorem is left as a side condition that should be proven before the test cases can be considered valid. This results in the following theorem:

$$\forall x, y : \mathbb{R} \bullet x \geq y \Leftrightarrow$$
$$(x = y) \lor (y < x \leq y + 0.1) \lor (x > y + 0.1)$$
$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$((n? = 0) \lor (0 < n? \leq 0 + 0.1) \lor (n? > 0 + 0.1))$$

The side condition is proven (e.g. using a predetermined proof tactic) and the existential quantifier simplified to leave the following three test cases.

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$n? = 0$$

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$0 < n? \leq 0.1$$

$$\vdash \exists n?, r! : \mathbb{R} \bullet n? \geq 0 \land r! \times r! = n? \land$$
$$n? > 0.1$$

Once the test partitions have been produced, satisfying test data can be generated by "solving" the existential quantifications using the constraint solvers in CADiZ. The partitioning method described above supports work by Stocks and Carrington [17, 18][3] who proposed a framework for the derivation and specification of test cases based on the Z notation (the Test Template Framework). The method of test case derivation described here complements that work by providing a mechanism for automatically applying the test heuristics to reveal the test partitions that can then be structured using the Test Template Framework.

## 3.2 Validation of Automated Testing Strategies

Mutation testing [7] is a fault-based testing technique that deliberately injects faults into a program in order to assess a test set's adequacy at detecting those faults. Based on the number of injected faults detected (mutation score), conclusions about the general fault finding ability (mutation adequacy) of the test set can be formed. Mutation testing provides a means of validating the test strategies discussed in the previous section. Automatic test case generation, as described above, can provide a statistically significant number of test cases for various strategies. The mutation adequacy of each of these strategies can then be assessed to compare their relative effectiveness at detecting faults [1].

Expressing a test case specification as a formal specification from which the test data are generated also opens up the possibility for some additional manipulation to increase the mutation score of the data. Mutation techniques can be applied at the specification level to create specifications that represent an abstract description of potential faults in the implementation (as first suggested by Budd and Gopal [4]). If test data can be generated from the original specification that identify (kill) the mutants, that data is also likely to achieve a relatively high score at the program level. The data generated from the specification would have been "hardened" in some sense against

[2] This can be roughly interpreted as: there exist some values for $n?$ and $r!$ that satisfy the specification and can therefore be used as suitable test data.

[3] As well as building on other important work in the area such as [11, 14, 8].

the likelihood of encountering co-incidental correctness in the implementation.

In general, the number of mutants that can be generated for an expressive formal specification notation such as Z would be extremely large. However, in practice, only a subset of the notation would be used for any particular application domain. In this case, the number of possible mutants would be limited. From within this subset more selective choices of which mutation strategies to apply can be made by analysing the mutation score of particular testing strategies. Hardened test data can then be generated from the test cases by strengthening the predicate of the test case to improve the probability that data are generated to kill the chosen set of specification mutants. In some cases, one set of test data could be generated to kill a number of mutants. However, where the hardening predicates are inconsistent, several sets of test data may need to be generated. Take as an example, the following simple test case for a system which averages two numbers:

$$\exists\, A, B, Result : \mathbb{N} \bullet Result = (A + B) div 2$$

The test data can be hardened against the mutation where the $+$ is replaced by a $-$ by adding an inequality to the test case.

$$\exists\, A, B, Result : \mathbb{N} \mid (A + B) \neq (A - B) \bullet Result = (A + B) div 2$$

The hardening predicate in this case was $(A + B) \neq (A - B)$. This represents a necessary condition for detecting the mutant but, in general, will not always be sufficient. Depending on other mutations that may arise elsewhere in the implementation, the new test case can not be guaranteed to produce data which kills the mutant, but is more likely to do so than without the hardening predicate. Mutation analysis was briefly mentioned by Stocks and Carrington [17] as an alternative testing heuristic to domain propagation in their Test Template Framework. However, the authors believe there is still scope for research in investigating effective mutation strategies for Z-based test sets and whether mutation analysis can be combined with standard domain partitioning to provide more effective test sets. Therefore, future work will evaluate various criteria for designing the hardening predicates and their relative efficacy at increasing mutation scores when applying the tests to the implementation. If hardening predicates could be automatically generated based on a known set of specification mutations, it may be possible to use the feedback from traditional mutation testing approaches (for assessing the effectiveness of test sets) to automatically select the most effective test strategies for particular types of program.

## 3.3 Proof as a Testing Oracle

Test data generated from formal specifications are typically not at the same level of abstraction as is needed to test the implementation. Some refinement will be needed to exercise the implementation with the test inputs. For implementations which do not preserve the structure of the original specification this refinement may be difficult. In addition, some specifications may be non-deterministic, eliminating the possibility of precalculating expected test results.

An alternative to the structured decomposition of the specification into test cases and expected results, as discussed above, is to use a "generate and test" approach. Test inputs are chosen via any means (e.g. randomly) and the results of applying the inputs to the implementation are then checked for conformance with the specification. This approach can also be used in conjunction with the partition-based testing. A statistically significant number of samples can be chosen from each test partition to increase the confidence in the equivalence class hypothesis used to generate the test cases. In either case, the process of checking the test inputs and outputs against the specification requires a test "oracle". If enough refinement information is known to transform the concrete inputs and outputs of the system into their equivalent in the abstract specification, the formal specification and automated proof tactics can be exploited to form an automated oracle. The specification is instantiated with the inputs and outputs and a proof tactic is used to simplify the expression to *True* (test passed) or *False* (test failed). Such simplification is ideally suited to automated theorem provers as it typically involves applying many "one-point" simplifications until the expression is reduced to either *True* or *False*.

# 4 Conclusions

In this paper we have shown how judicious use of testing and proof to support one another can lead to significant benefits for the software V&V process, both in terms of increased automation and integrity. The use of counter-example generation can save much wasted proof effort and the use of proof to support test case design can be used to demonstrate the correctness of the test partitioning techniques as well as offering a means of automation in itself.

The high level of automation is made possible because of the combination of restricting the subset of the formal notation used, the ability to predict the structure of the proofs that are required (and therefore the ability to re-use proof tactics many times) and the use of a powerful theorem proving tool with integrated constraint solving abilities. In the authors' experience in aerospace applications, these restrictions did not need to be contrived but occurred naturally as a property of the domain and the types of proof that were performed.

Some of the techniques described here (e.g. counter-example generation and automated test case and data generation) have already been applied to a large industrial case study [5]. Other techniques, (e.g. automated proof as a testing oracle and application of mutation testing concepts) require more research to fully explore their potential. In particular, the use of mutation testing techniques, both at the code and specification level appears a promising method of automatically generating effective and efficient test criteria for Z-based testing of particular application domains. This is an area of research that is made possible by the automated framework described in this paper and will be the focus of future work.

# 5 Acknowledgements

This work was funded by the High Integrity Systems and Software Centre, Rolls-Royce Plc.

# References

[1] S.P. Allen and M.R. Woodward. Assessing the quality of specification-based testing. In Sandro Bologna and Giacomo Bucci, editors, *Proceedings of the third international conference on achieving quality in software*, pages 341–354. Chapman and Hall, 1996.

[2] Boris Beizer. *Software Testing Techniques*. Thomson Computer Press, 1990.

[3] Sergey Berezin. The SMV web site. http://www.cs.cmu.edu/~modelcheck/smv.html/, 1999. The latest version of SMV and its documentation may be downloaded from this site.

[4] Timothy A. Budd and Ajei S. Gopal. Program testing by specification mutation. *Computer Languages*, 10(1):63–73, 1985.

[5] Simon Burton, John Clark, Andy Galloway, and John McDermid. Automated V&V for high integrity systems, a targeted formal methods approach. In *Proceedings of the 5th NASA Langley Formal Methods Workshop*, June 2000.

[6] John Clark and Nigel Tracey. Solving constraints in LAW. LAW/D5.1.1(E), European Commission - DG III Industry, 1997. Legacy Assessment WorkBench Feasibility Assessment.

[7] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, pages 34–41, April 1978.

[8] J Dick and A Faivre. Automating the generation and sequencing of test cases from model-based specifications. *FME'93:Industrial Strength Formal Methods, Europe. LCNS 670*, pages 268–284, April 1993.

[9] John B Goodenough and Susan L Gerhart. Towards a theory of test data selection. *IEEE Transactions On Software Engineering*, 1(2):156–173, June 1975.

[10] The VDM-SL Tool Group. *The IFAD VDM-SL Language*. The Institute of Applied Computer Science, September 1994.

[11] P A V Hall. Towards testing with respect to formal specifications. *Second IEE/BCS Conference On Software Engineering*, pages 159–163, 1988.

[12] Daniel Jackson and Craig Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Transactions on software engineering*, 22(7):484–495, July 1996.

[13] A. K. Mackworth. Consistency in networks of relations. *Artifical Intellegence*, 8:99–118, 1977.

[14] Thomas J Ostrand and Marc J Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, June 1988.

[15] Martyn Ould. Testing - a challenge to method and tool developers. *Software Engineering Journal*, 39:59–64, March 1991.

[16] J. M. Spivey. *The Z Notation: A Reference Manual, second edition*. Prentice Hall, 1992.

[17] Phil Stocks and David Carrington. Test template framework: A specification-based case study. *Proceedings Of The International Symposium On Software Testing And Analysis (ISSTA'93)*, pages 11–18, 1993.

[18] Phil Stocks and David Carrington. A framework for specification-based testing. *IEEE Transactions On Software Engineering*, 22(11):777–793, November 1996.

[19] I. Toyn. Formal reasoning in the Z notation using CADiℤ. *2nd International Workshop on User Interface Design for Theorem Proving Systems*, July 1996.

[20] Ian Toyn. A tactic language for reasoning about Z specifications. In *Proceedings of the Third Northern Formal Methods Workshop, Ilkley, UK*, September 1998.

# Test Generation and Recognition with Formal Methods

Paul E. Ammann
George Mason University
Information & Software Eng. Dept.
Fairfax, Virginia 22033 USA
pammann@gmu.edu

Paul E. Black
National Institute of Standards and Technology
100 Bureau Dr., Stop 8970
Gaithersburg, Maryland 20899 USA
paul.black@nist.gov

## Abstract

*The authors are part of a larger group at the National Institute of Standards and Technology (NIST), George Mason University (GMU), and the University of Maryland, Baltimore County (UMBC). Projects directed by group members use formal methods, particularly model checking, to investigate the generation and recognition of test sets for software systems. Our positions, in order of increasing potential controversy, are 1) the use of specifications is an important complement to code-based methods, 2) test set recognition is as important as test set generation, and 3) in spite of some known limitations, our generic framework for testing, with a test criterion as a parameter and a model checker for an engine, is a general approach that can handle many interesting specification-based test criteria.*

## 1. Our Relevant Work

For the context of our position, we summarize our recent contributions to specification-based testing using a model checker. Model checkers, which evaluate finite state machines with respect to temporal logic constraints, are chosen in favor of theorem proving approaches because 1) significantly less expertise is required of the end user, thereby enhancing automation, 2) model checkers are enjoying an explosive growth in applicability, and 3) the counterexamples from a model checker may be directly interpreted as test cases.

In our original paper on the topic [4], we defined mutation testing for model checking specifications, specifically SMV descriptions. We defined one class of mutation operators that changed the state machine description; these operators result in failing tests, that is, tests that a correct implementation must reject. We defined another class of mutation operators that changed the temporal logic constraints on the state machine; these operators result in passing tests, that is, tests that a correct implementation must accept. The model checker identifies equivalent mutants: these are temporal logic constraints that are consistent. We generated tests for a small example, ran them against a target implementation, and measured code branch coverage.

Generating tests to "kill" all mutants is the first test criterion we investigated. Some other specification-based criteria are stuck-at faults [1], CCC partitions [6], MC/DC [7], automata theoretic [8], branch coverage [10], disconnection or redirection faults [11], and transition pair coverage [13]. Test generation then is the problem of finding tests which fulfill the goals embodied in the criterion. Test set recognition is the conjugate of test generation. Whereas test generation asks, "What tests will satisfy the test criteria?", test recognition asks, "How much of the test criteria do these tests satisfy?"

In follow-on work [3], we addressed test set recognition for a refinement of the mutation analysis scheme. In particular, we defined a metric in terms of number of mutants killed by a given test set compared to the total number of killable mutants. We showed how to turn tests from a candidate test set into "forced" state machines and then use the model checker to compute the metric. We analyzed various factors that could introduce distortions, such as semantically equivalent mutants and mutants that are killed by every test case, were analyzed.

We analyzed different mutation operators both theoretically and empirically [5]. For theoretical analysis, we applied predicate differencing and a hierarchy of fault classes [12]. To experimentally confirm the conclusions, we generated tests using many mutation operators for three different small examples and compared relative coverage of the different operators. Although mutation operators do not correspond exactly to fault classes, we found good correlation between them. We defined a composite mutation operator which gave the maximum coverage, and found a single mutation operator which gave nearly-maximum coverage using far fewer mutants.

Although the above methods work well for state machine specifications, most specifications are written at higher levels in Z, UML, OCL, SCR, etc. So to be practical, there must be (semi-)automatic ways of extracting simpler pieces which can be analyzed. In [2] we defined a new algorithm to abstract a simple state machine, focusing on some states of interest to an analyst, from an unbounded description. We proved that the algorithm is sound for test generation. That is, any test produced corresponds to a passing tests in the original unbounded description.

We also applied the work to the problem of network security [14], particularly cases where configuration changes on one machine can lead to vulnerabilities on other machines in a network. Network configurations were encoded as a state machine, along with the transformations produced by known attacks. Security policies are stated in the temporal logic in forms such as "Under a set of assumptions, someone outside the firewall cannot obtain root access on machine X." If the configuration in fact allows such access given the set of known attacks, a counterexample is produced illustrating the attack.

In work underway, we encoded different test criteria as temporal logic constraints. Using a model checker, we analyzed branch coverage [10], uncorrelated full-predicate coverage (similar to Multiple Condition/Decision Coverage or MC/DC [7]), and transition-pair coverage [13], in addition to mutation coverage. We found that different metrics are easily encoded into temporal logic, with some limitations, and that interesting theoretical comparisons between metrics are facilitated by formalizing them.

To scale these methods up to problems of useful size and general nature, we successfully applied them to several different examples. We began with small, well-known examples such as Cruise Control and Safety Injection. We also modeled the operand stack of a Java virtual machine and several functional source code benchmarks for unit testing, then generated good test sets. Currently we are applying the method to a part of a flight guidance system from an aerospace firm and to a secure operating system add-on for a Unix derivative.

## Other Work

The earliest work we know of on generating tests using model checkers is when Callahan, Schneider, and Easterbrook [6] mentioned that counterexamples generated from SPIN, MurΦ, or SMV model checkers can be used as test cases.

Engels, Feijs, and Mauw [9] named some general concepts, such as "test purposes" (some goals to achieve with testing) and "never-claim" (submit the negation of what you want so the model checker finds a positive instance). They discussed positive and negative testing. Positive testing checks that the system does what it should, which is appropriate for general system checks. Negative testing looks for a particular action the system should *not* do. The disadvantage is that one must specify the errors to look for, but it may be useful in searching for particular errors.

Most recently Gargantini and Heitmeyer [10] developed a requirements branch or case coverage test purpose using the SPIN or SMV model checkers. Also conditions in requirements may be elaborated in the test purposes to exercise boundary conditions, for instance, $x \geq y$ may be split into $x > y$ and $x = y$.

## 2. Research Questions

In January 2000 the group held an informal workshop at NIST. Some 20 scientists, professors, and students spent half a day sharing their views on the work, listing programs we need, and defining research topics and questions, such as:

1. What are the effects of semantically identical, but syntactically different specification styles on test set quality?

2. How do we make tests observable?

3. How can we partition a huge model between light- and heavy-weight formal methods, then combine their results to get tests?

4. What are the advantages and disadvantages for test generation or expressibility with SMV and SPIN (CTL vs. LTL)?

5. How can (should) we trade off number of tests and coverage?

6. What are good (semi-)automatic abstractions from large, even infinite descriptions for test generation?

7. Can we use state machine mutations (failing tests) to check systems for safety?

8. What are a good set of mutation operators, e.g., for larger models.

9. How do duplicate mutants affect coverage metrics? Do some sets of mutation operators produce many or few duplicates?

## 3. Position Statement

- The use of specifications is an important complement to code-based methods.

This is an old position, but we argue that recent trends in software development and testing make it more compelling. The traditional argument, which is still valid, is that without a specification, we do not know to test for features which are entirely missing from the source code. More importantly, in acceptance tests of binary programs or conformance testing without a reference implementation, there is no source code available at all. During rapid development it may be helpful to write tests in parallel with or even preceding coding; such a model is directly supported by the use of "use-cases" in requirements analysis. Use-cases are essentially system tests, and analyzing use-cases with respect to specification-based test metrics is an important research area. Further, there is a body of research that aims to introduce formal methods into industrial development by amortizing the cost of developing formal specifications over other, traditionally expensive, life-cycle phases, particularly testing. Model-checkers are a relatively new, but powerful tool in achieving this objective.

- Test set recognition is as important as test set generation.

There are basically two thrusts to this argument, one theoretical and the other practical. The theoretical argument is that scientific comparisons between test methods benefit greatly if a test set produced by method A can be evaluated directly and without bias with respect to method B. Test methods that focus purely on test generation do not satisfy this objective. The practical argument is that industry has an enormous investment in existing test sets, primarily in regression test sets, but also in new development artifacts such as use-cases from requirement analysis. To retain the value of this investment, it is much more helpful to critique the existing artifacts with statements of the form, "Tests of type X and Y are missing," rather than merely providing a new test set that bears no relation to the existing ones.

- In spite of known limitations, our generic framework for testing, with a test criterion as a parameter and a model checker for an engine, is a general approach that can handle many interesting specification-based test criteria.

We define a model whereby a test criterion is paired with a specification of a specific application, and, with as much automation as possible, test requirements specific to the application are generated and satisfied with specific tests, either new or old. The key is the degree of automation. We believe that using a temporal logic to express the test requirements and a model checker to create and/or match test cases to test requirements is a general purpose approach suitable for many specification-based test methods. As described above, this approach has been successful for a variety of interesting test criteria. One interesting aspect of this line of research has been in discovering where the method falls short. The significant result so far is that any test requirement that places constraints on pairs of tests (as opposed to individual tests) is not well handled by a model checker, since counterexamples are typically generated one at a time. An example is the

MC/DC metric, popular in avionic applications. In MC/DC, pairs of tests are required to differ in the value of exactly one condition. The research question is how to work around this expressibility constraint.

## References


[1] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital System Testing and Testable Design*. IEEE Computer Society Press, New York, N.Y., 1990.

[2] Paul Ammann and Paul E. Black. Abstracting formal specifications to generate software tests via model checking. In *Proceedings of the 18th Digital Avionics Systems Conference (DASC99)*, volume 2, page 10.A.6. IEEE, October 1999. Also NIST IR 6405.

[3] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. In *Proceedings of Fourth IEEE International High-Assurance Systems Engineering Symposium (HASE 99)*, pages 239–248. IEEE Computer Society, November 1999. Also NIST IR 6403.

[4] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proceedings of the Second IEEE International Conference on Formal Engineering Methods (ICFEM'98)*, pages 46–54. IEEE Computer Society, December 1998.

[5] Paul E. Black, Vadim Okun, and Yaacov Yesha. Mutation operators for specifications. In $15^{th}$ *IEEE International Conference on Automated Software Engineering (ASE2000)*, October 2000. Submitted.

[6] John Callahan, Francis Schneider, and Steve Easterbrook. Automated software testing using model-checking. In *Proceedings 1996 SPIN Workshop*, Rutgers, NJ, August 1996. Also WVU Technical Report #NASA-IVV-96-022.

[7] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, pages 193–200, September 1994.

[8] Tsun S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, SE-4(3):178–187, May 1978.

[9] André Engels, Loe Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In Ed Brinksma, editor, *Proceedings of the Third International Workshop on Tools and Algorithms for the Construction and Analysis of Systems. (TACAS'97)*, volume 1217 of *Lecture Notes in Computer Science*, pages 384–398. Springer-Verlag, April 1997.

[10] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proceedings of the Joint 7th European Software Engineering Conference and 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Toulouse, France, September 1999. To Appear.

[11] Jens Chr. Godskesen. Fault models for embedded systems. In *Proceedings of CHARME'99*, volume 1703 of *Lecture Notes in Computer Science*. Springer-Verlag, September 1999.

[12] D. Richard Kuhn. Fault classes and error detection in specification based testing. *ACM Transactions on Software Engineering Methodology*, 8(4), October 1999.

[13] Jeff Offutt, Yiwei Xiong, and Shaoying Liu. Criteria for generating specification-based tests. In *Proceedings of the Fifth IEEE Fifth International Conference on Engineering of Complex Computer Systems (ICECCS '99)*, pages 119–131, Las Vegas, NV, October 1999. IEEE Computer Society Press.

[14] Ronald W. Ritchey and Paul Ammann. Using model checking to analyze network vulnerabilities. In *Proceedings 2000 IEEE Computer Society Symposium on Security and Privacy*, Oakland, CA, May 2000. To Appear.

# A Formally Founded Componentware Testing Methodology

Klaus Bergner, Heiko Lötzbeyer, Andreas Rausch
Marc Sihling, Alexander Vilbig

Institut für Informatik
Technische Universität München
80290 Munich, Germany

## Abstract

In this position paper we propose an overall methodology for specification-based testing that is founded on a formal model of component systems. We motivate the importance of clearly defined description techniques and cover their role with respect to techniques for the generation and validation of test cases.

## 1 Introduction

Traditionally, the role of formal development techniques in system testing has been rather weak, partly due to the wrong perception that the use of formal techniques can abolish the need for testing. However, experience has shown that this assumption does not hold in practice, as the effort for the formal verification of large real systems is not manageable in most cases. In this paper, we want to show that formal techniques can and should be used not to abolish, but to supplement the established testing methodologies.

Testing is an established discipline in software engineering, and there exists a variety of tools, techniques, and process models in this area [Bei90, Gri92, Kit95]. Most informal methodologies rely on the creation of two completely separated models: the design specification is used to describe the system's functionality and architecture, while the test specification characterizes the set of corresponding test cases. At the one hand, this separation is necessary to uncover omissions and errors not only in the code of a system, but also in its design specification. At the other hand, much effort must be put into assuring the consistency of the two models in order to make tests possible and meaningful. Furthermore, the informal approach seems to reach its limits with large distributed component systems. Here, the absence of a global state makes it very difficult to specify and perform reasonably complete, reproducable tests without relying strongly on the system's design specification.

Most specification-based methods try to resolve these disadvantages by generating code and test cases from a common formal design specification [Ulr98]. However, this approach gives up the decisive advantage of separating both specifications—according to it, only specified properties of the system can be tested.

In this position paper, we propose a more general approach: Both system specification and test specification should be based on a common formal model with clear definitions and consistency conditions for design concepts as well as testing concepts. Thus, developers can use formally founded description techniques to specify the component system and its test cases, both derived from the informal user requirements. Techniques and tools based on the formal model allow to check the consistency of the test model with the design model, even if they both have been created more or less independently. Furthermore, the approach offers the possibility of generating test cases both from the design specification as well as from the test specification.

In the following section, we first explain our vision of specification-based testing and clarify the base concepts. In Section 3, we then cover the role of formally founded description techniques, especially with respect to integration testing. Section 4 deals with techniques and methods for the use of such description techniques, focusing on the generation and validation of test cases. A short conclusion with an outlook ends the paper.

## 2 Specification-Based Testing

In our view, a universal componentware development methodology consists of four basic constituents (cf. [BRSV98c]). Testing plays an important role with respect to all of them:

**System Model:** The formal system model represents the foundation of the methodology. It defines the essential concepts of componentware, including mathematical definitions for the notion of a component, an interface, and their behavior. Also contained are definitions for testing concepts like test runs or test

results [BRS+ar].

**Description Techiques:** The formal system model is not intended to be used for development directly, as using it requires experience with formal methods. For developers, a set of intuitive graphical and textual description techniques is provided in order to describe and specify the system and its components. This pertains not only to design descriptions of the system itself, but also to test case specifications and the corresponding consistency conditions [HRR98, BRSV98c].

**Process Model:** In order to apply the description techniques in a methodical fashion, a development and testing process has to be defined. In the small scale, this pertains to single techniques like the transformation and refinement of diagrams or the generation of test cases from design specifications. In the large scale, the development and testing activities of developers, testers, and managers in different roles have to be coordinated. This includes, for example, the definition of new roles like black box testers for commercial components, or test case designers for distributed integration tests [BRSV98a, BRSV98b, ABD+99].

**Tools:** At least, tools should support the creation of description techniques. Furthermore, they should be able to generate part of the code, the documentation, and the test cases for the system. Beyond that, many applications are possible, ranging from consistency checks over simulation to development workflow support [HSSS96].

Based on former work on system models and formal description techniques [Bro95, KRB96, GKR96, BHH+97], we have presented a formal componentware system model in [BRS+ar], yet without explicit support for test concepts. The base concepts of the model in its current form are as follows:

- *Instances* represent the individual operational units of a component system that determine its overall behavior. With componentware, this pertains to component, interface, and connection instances, and their various relations and properties. Each component instance has a defined behavior, determining its interaction history with respect to incoming and outgoing messages and to structural changes. As the whole component system can be seen as a component itself, the state and interaction history of whole component systems can also be captured. This includes not only interactions between communicating components, but also the structural behavior of the system, understood as the changes to its connection structure and the creation and deletion of instances at runtime.
- *Types* address disjoint subsets of interface resp. component instances with similar properties. Each instance is associated to exactly one type. Component types are used to capture component instances with a common behavior.
- *Descriptions* are assigned to types in order to characterize the behavior of their instances. Our notion of a description is very wide: examples are interface and component signatures, state transition diagrams, extended event traces, or even source code. For each description, there exists an interpretation which translates the notation into terms of the formal system model. Each description can thus be represented by a predicate which checks whether certain properties of the system are fulfilled. Based on this clear, formal understanding, consistency conditions between the different description techniques can be defined.

Ideally, all descriptions should hold for all described instances. In practice, however, only few descriptions may be checked or enforced statically—apart from simple interface signatures, this requires formally founded description techniques with refinement and proof calculi that allow the verification or generation of code. In practice, one mostly has to resort to extensive testing.

In the context of the outlined methodology and the formal system model, the testing approach proposed in the introduction can now be presented in more detail. According to our vision for an overall specification-based componentware testing methodology, developers elaborate two different specifications based on the initial, informal customer requirements: the formal system specification as well as the formal test specification. Both are specified with intuitive graphical and textual description techniques based on the common formal system model. Developers are provided a variety of techniques and tools, for example, for checking the consistency between the two models, for generating test cases from design and test descriptions, and for validating the consistency of manually created test cases with the design descriptions (cf. Section 4).

Once all specifications have been elaborated to some extent, testing itself can start. First, the initial system state specified in the respective test case has to be established by creating the corresponding component instances, setting their state, and creating the connections between them. During the test run, the external stimuli described in the test case have to be executed on the component instances, and the resulting communication and structural behavior must be checked for compliance with the specification given in the test case. At the end of the test run, the final state of each component instance and the connection structure of the system has to be compared with the desired state and connection structure specified in the test case.

In the next section, we make some remarks about suitable description techniques, especially with regard to the repre-

sentation of integration test case descriptions.

# 3 Description Techniques for Integration Testing

As told in the previous section, there exists a variety of different description techniques for various aspects of a system. We want to base our work on the techniques provided by the UML [Gro99], adapting and refining them when necessary. With respect to integration testing, the following kinds of graphical description techniques are especially interesting:

- State-based descriptions, like *state transition diagrams* based on input/output automata are well suited to describe the state changes and the communication behavior of components or (sub)systems.
- Structural description techniques, like *instance diagrams* can be used to describe the connection structure of a component system.
- Interaction-based descriptions, like *sequence diagrams*, can be used to describe exemplary or desired interaction sequences of the components in a system.

Based on these three graphical description techniques, *test case descriptions* can be composed. A typical test case may contain:

- A specification of an *initial configuration*, described by an instance diagram and state diagrams for the participating components.
- A specification of the *test case behavior*, pertaining to the communication of the considered components with each other and with the test environment as well as to their structural behavior. This specification contains sequence diagrams or state transition diagrams specifying the external stimuli that have to be initiated by the test environment during the test run as well as the expected output reactions of the test object. With respect to the structural changes, instance diagrams can be used to specify desired intermediate configurations.
- A specification of the desired *terminal configuration*, similar to the specification of the initial configuration.

While this overall approach seems to be viable, there remain many open questions, for example, with respect to the adequate syntactical representation of test cases, the exact semantics of state transition diagrams and sequence diagrams, and the treatment of nondeterministic descriptions. Another question arises with instance diagrams: like most other structural description techniques, they can only capture snapshots of a system configuration at a certain time, making them unsuitable for the description of the behavior of large dynamic systems.
The following steps have to be done in order to develop a toolkit of description techniques suitable for integration testing:

1. Identification of requirements for description techniques that are suited for the description of the behavior of a distributed component system.
2. Elaboration of a toolkit of formally founded description techniques.
3. Definition of integration test case descriptions based on the toolkit of description techniques.

# 4 Methods and Techniques

The development of a successful test design is both an extensive and difficult task. On the one hand, a large amount of test data has to be generated and managed. On the other hand, each test case should be of high quality in order to maximize the probability of finding remaining faults.
In the context of an overall integration testing methodology based on graphical descriptions, at least the following testing activities should be supported:

- Generation of test cases from design and test specifications.
- Consistency checking of manually created test cases with design specifications.
- Automated execution of test cases and evaluation of test results.
- Visualization of test results and animation of test runs based on graphical descriptions.
- Analysis of test coverage and likelihood of error detection.
- Management, organization, and documentation of test results.

As the manual creation of test cases is a rather tedious and error-prone process that requires much effort, we want to focus on the automatic generation and validation of test cases. As has been shown, automating test case generation can make testing easier and more effective [JPP+97]. Several techniques for automatic test case generation from behavior descriptions like Mealy Machines [Cho78, ADLU91, FvBK+91, Ura92], X-Machines [IH98], VDM [DF93], or Z [Sad99, HNS97] have been developed in the past. Recently, these techniques have been applied to current industrial graphical description techniques like ones provided by UML [Gro99]. Unfortunately, the new approaches consider only single aspects of UML and the

testing process. For example, [KHBC99] presents a method for generating test cases from UML state diagrams. As the scope of the discussed method is clearly restricted to unit testing, further work is is needed for integration testing.
Furthermore, other approaches used for test case generation and validation should be incorporated into the testing process. We propose, for example, the use of the classification tree method developed by Grimm [Gri95]. This method is particularly well-suited for choosing concrete input data by classifying the input data space. In addition, techniques based on model checking can complement the testing process by generating additional test cases from system properties [EFM97] or by validating test cases derived from a test specification [NS93]. We believe that a powerful testing methodology should provide a toolkit of techniques for test case generation and validation.
Therefore, in parallel to the development of suitable description techniques (cf. Section 3), the following steps have to be performed for the development of an adequate testing methodology:

1. Analysis and evaluation of existing algorithms with respect to the covered description techniques and the special needs of integration testing.
2. Analysis and evaluation of complementary generation and validation techniques for integration testing.
3. Development of test case generation and validation techniques for high-level graphical description techniques.
4. Integration of selected techniques into a consistent toolkit.

# 5 Conclusion and Outlook

The research agenda as given at the end of Sections 3 and 4 represents only the first step on the way to a practical specification-based testing methodology. After the conceptual foundation has been developed, the techniques should be validated on a small application example, for example, in the context of a distributed CORBA system.
Further work is then needed to transfer the method into practice. First, a suitable test infrastructure for the execution of test cases must be developed. In the following, tool support for the generation of test cases from graphical description techniques has to be provided, for example, by extending an existing CASE tool.

# Acknowledgements

We thank Andreas W. Ulrich for enlightening discussions and comments on earlier versions of this paper.

# References

[ABD+99] Dirk Ansorge, Klaus Bergner, Bernd Deifel, Nicholas Hawlitzky, Andreas Rausch, Marc Sihling, Veronika Thurner, and Sascha Vogel. Managing componentware development – software reuse and the V-Modell process. In *Proceedings of CAiSE '99*, Lecture Notes in Computer Science. Springer-Verlag, 1999.

[ADLU91] A. V. Aho, A. T. Dahbura, D. Lee, and M. U. Uyar. An optimization technique for protocol conformance test generation based on UIO sequences and rural chinese postman tours. *IEEE Transactions on Communications*, 39(11):1604–1615, 1991. An earlier version with a same title appeared in *Proc. of the IFIP WG 6.18th International Symposium on Protocol Specification, Testing, and Verification, June 1988*.

[Bei90] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, 2nd edition, 1990.

[BHH+97] R. Breu, U. Hinkel, C. Hofmann, C. Klein, B. Paech, B. Rumpe, and V. Thurner. Towards a formalization of the unified modeling language. In *Proceedings of ECOOP'97*. Springer Verlag, LNCS, 1997.

[Bro95] Manfred Broy. Mathematical system models as a basis of software engineering. *Computer Science Today*, 1995.

[BRS+ar] Klaus Bergner, Andreas Rausch, Marc Sihling, Alexander Vilbig, and Manfred Broy. A formal model for componentware. In Murali Sitaraman and Gary T. Leavens, editors, *Foundations of Component-Based Systems.* Cambridge University Press, 1999 (to appear).

[BRSV98a] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. Technical Report I-9823, Technische Universität München, Institut für Informatik, 1998.

[BRSV98b] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. A componentware development methodology based on process patterns. In *PLOP'98 Proceedings of the 5th Annual Conference on the Pattern Languaes of Programs*. Robert Allerton Park and Conference Center, 1998.

[BRSV98c] Klaus Bergner, Andreas Rausch, Marc Sihling, and Alexander Vilbig. An integrated

view on componentware - concepts, description techniques, and development process. In Roger Lee, editor, *Software Engineering : Proceedings of the IASTED Conference '98*. ACTA Press, Anaheim, 1998.

[Cho78] T. S. Chow. Testing Software Design Modeled by Finite-State Machines. *IEEE trans. on Software Engineering*, SE-4, 3:178–187, 1978.

[DF93] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In J. C. P. Woodcock and P. G. Larsen, editors, *FME'93: Industrial-Strength Formal Methods*, pages 268–284. Formal Methods Europe, Springer-Verlag, April 1993. Lecture Notes in Computer Science 670.

[EFM97] A. Engels, L. Feijs, and S. Mauw. Test generation for intelligent networks using model checking. *Lecture Notes in Computer Science*, 1217:384–??, 1997.

[FvBK+91] S. Fujiwara, G. von Bochmann, F. Khendek, M. Amalou, and A. Ghedamsi. Test Selection Based on Finite-State Models. *IEEE Transactions on Software Engineering*, 17(6):591–603, June 1991.

[GKR96] Radu Grosu, Cornel Klein, and Bernhard Rumpe. Enhancing the SysLab system model with state. Technical Report TUM-I9631, Technische Universität München, Institut für Informatik, 1996.

[Gri92] K. Grimm. Systematisches Testen sicherheitsrelevanter Software – Methoden, Verfahren und Werkzeuge. *Informatik zwischen Wissenschaft und Gesellschaft. Informatik Fachberichte 309*, pages 66–107, 1992.

[Gri95] K. Grimm. Systematisches Testen von Software - Eine neue Methode und eine effektive Teststrategie. Technical report, GMD-Forschungszentrum Informationstechnik GmbH, München/Wien, 1995.

[Gro99] UML Group. Unified Modeling Language. Version 1.3, Rational Software Corporation, 1999.

[HNS97] S. Helke, T. Neustupny, and T. Santen. Automating test case generation from Z specifications with Isabelle. *Lecture Notes in Computer Science*, 1212:52–??, 1997.

[HRR98] Franz Huber, Andreas Rausch, and Bernhard Rumpe. Modeling dynamic component interfaces. In *Proceedings of TOOLS'98, to appear*, 1998.

[HSSS96] Franz Huber, Bernhard Schätz, Alexander Schmidt, and Katharina Spies. Autofocus - a tool for distributed systems specification. In Bengt Jonsson and Joachim Parrow, editors, *Proceedings FTRTFT'96 - Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 467–470. LNCS 1135, Springer Verlag, 1996.

[IH98] F. Ipate and M. Holcombe. A method for defining and testing generalised machine specification. *Int. Jour. Comp. Math.*, 68:197–219, 1998.

[JPP+97] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L. G. Votta. Specification-based Testing of Reactive Software: Tools and Experiments. In *Proceedings of the International Conference on software Engineering*, 1997.

[KHBC99] Y. G. Kim, H. S. Hong, D. H. Bae, and S. D. Cha. Test cases generation from UML state diagrams. In *IEE Proceedings: Software*, 146(4), 187-192, August 1999.

[Kit95] Edward Kit. *Software Testing in the Real World: Improving the Process*. Addison Wesley, Wokingham, 1995.

[KRB96] Cornel Klein, Bernhard Rumpe, and Manfred Broy. A stream-based mathematical model for distributed information processing systems: The SysLab system model. In J.-B. Stefani E. Naijm, editor, *FMOODS'96 Formal Methods for Open Object-based Distributed Systems*, pages 323–338. ENST France Telecom, 1996.

[NS93] K. Naik and B. Sarikaya. Test case verification by model checking. *Formal Methods in System Design*, 2:277–321, 1993.

[Sad99] Sadegh Sadeghipour. *Testing Cyclic Software Components of Reactive Systems on the Basis of Formal Specifications*. Verlag Dr. Kovac, 1999.

[Ulr98] A. W. Ulrich. *Testfallableitung und Testrealisierung in verteilten Systemen*. Shaker Verlag, Aachen, 1998.

[Ura92] H. Ural. Formal methods for test sequence generation. *Computer communications*, 15(5), June 1992.

# Dynamic Analysis

# Java Model Checking

David Y.W. Park  Ulrich Stern  Jens U. Skakkebæk*
David L. Dill

*Computer Science Department*
*Stanford University*
*Stanford, CA 94305*

{parkit,uli,jus,dill}@cs.stanford.edu

## Abstract

This paper presents initial results in model checking multi-threaded Java programs. Java programs are translated into the SAL (Symbolic Analysis Laboratory) intermediate language, which supports dynamic constructs such as object instantiations and thread call stacks. The SAL model checker then exhaustively checks the program description for deadlocks and assertion failures. Basic model checking optimizations that help curb the state explosion problem have been implemented. To deal with large Java programs in practice, however, supplementary program analysis tools must work in conjunction with the model checker to make verification manageable. The SAL language framework provides a good starting point to interface new and existing analysis methods with the model checker.

## 1 Introduction

The Java programming language is becoming increasingly popular for writing multi-threaded applications. In particular, many Internet servers are written in Java. Since Java has multi-threading built in among other advantages, we expect it to gain popularity in other areas such as embedded systems where multi-threading is useful.

Developing multi-threaded programs is notoriously difficult, however. Subtle program errors can result from unforeseen interactions among multiple threads. In addition, these errors can be very hard to reproduce since they often depend on the non-deterministic behavior of the scheduler and the environment.

It is thus desirable to provide tools for software developers that automatically detect errors due to multi-threading. The tools should generate detailed error traces to help the developer during the debugging phase. We have developed such a tool based on *model checking*. A model checker explores all reachable states of a system model, checking whether they satisfy the user-provided correctness specification including the absence of deadlocks and assertion failures. Our tool verifies models described in the SAL (Symbolic Analysis Laboratory) intermediate representation [10], that Java, among other languages, can be translated down to.

Our research focus was in developing a framework that is tailored towards software verification. The SAL model checker supports dynamically changing data structures, which are used, for example, to model Java object creation and call stacks. Popular traditional model checkers like SPIN [12] or Mur$\varphi$ [5] support only constant-size data structures, although an extension of SPIN called dSPIN [4] has been recently developed to support dynamic data structures. Nonetheless, the SAL model checker and the broader SAL language framework was initially designed with software verification in mind, with the intention of providing a system that can deal efficiently with the dynamic aspects of software.

The main challenge in model checking is the *state explosion problem* – the number of states in the model is frequently so large that model checkers exceed the available memory and/or the available time. We have incorporated into our tool two tech-

*Jens Skakkebæk is now at Adomo Inc., Cupertino, CA.

niques to combat state explosion, a form of partial order reduction and hash compaction, and are in the process of implementing additional optimizations.

We also plan to integrate the SAL model checker with program analysis tools that prepare large programs for efficient verification. Our model checker, for instance, could serve as a back-end to the Bandera [2] framework, giving us direct access to Bandera's slicing and abstraction tools that may turn intractable verification models into tractable ones. To allow for easy integration with Bandera, we chose to use McGill's Jimple [17], a three-address representation of Java byte-code on which the Bandera tools operate, as the input language of our model checker.

Related works include the model checkers Java PathFinder [11] and JCAT [3], both of which translate Java into SPIN's input language PROMELA. Since SPIN does not support dynamic data structures, they have to allocate fixed-size heaps and stacks. In addition, both tools translate directly from Java source code rather than from byte-code. Hence, they cannot be easily integrated into Bandera nor verify programs where only the byte-code is available. Moreover, the translation process is more complicated since several advanced Java features like exceptions have a simpler byte-code representation than a source-code one. The Java PathFinder group at NASA is in the process of addressing many of these issues with a new version of their model checker.

Recently, there has been increasing interest in verification tools that rely on the execution of actual code, eliminating the need to represent program states and statements using a specialized description language. VeriSoft [9], for example, can detect errors in C-style concurrent programs by monitoring program execution and systematically directing the scheduler. More recently, Bruening has integrated a deterministic tester into the Rivet Virtual Machine at MIT, that can detect deadlock and assertion failures in Java programs [1]. The tool relies on checkpointing system states in the virtual machine to backtrack to previous states during testing, but like VeriSoft, does not store states that it has already visited. Hence, the disadvantage of these tools is that the same state may be visited multiple times. Consequently, large portions of the state space may be explored redundantly, and non-terminating programs (e.g., server-side processes that loop indefinitely) become problematic. The SAL model checker borrows ideas from VeriSoft-like tools, but couples them with model checking techniques to efficiently explore the state space. Furthermore, our model checker is written in C++ with run-time efficiency in mind.

After giving a high-level overview of the model checker in Section 2, details of the model checker and the methods to increase the size of the models that it can handle are explained in Section 3. Section 4 describes the translation steps from Jimple to an executable model checker. We give results on several Java sample programs in Section 5, and conclude in Section 6.

## 2 Overview of the Model Checker

The SAL model checker explores all reachable states of a given Java program. The two sources of nondeterminism during this state exploration are the choice of the next thread to run (scheduling) and the input values from the environment. The model checker currently detects deadlocks and assertion violations.

The model checker executable is generated in a sequence of translation steps that includes converting Jimple into the SAL intermediate language [10]. Using an intermediate representation has two advantages. First, languages other than Java can be translated into the intermediate representation, reducing the effort to develop a model checker for each new language. Second, other analysis tools that accept SAL as their input language can be readily used to analyze Java.

SAL is a language for describing transition systems. The transition system is described with a set of guarded commands, each of which consists of a boolean condition on the current state and an associated action that changes the current state into the next state. SAL has two slightly different forms – Level 1 and Level 0. While SAL Level 1 has explicit guarded commands, SAL Level 0 folds the guarded commands into one large transition function.

The SAL language has several features that make it a good target for software model checkers. First, SAL provides unbounded arrays whose sizes vary

dynamically. These arrays are used, for example, to hold dynamically created Java objects and threads. (A Java thread is an object of class `java.lang.Thread`.) Second, SAL has abstract data types that can be used as unions. A stack frame, for example, is modeled as a record containing, among other fields, a union that can hold values for each possible method return type.

The model checker executable is generated in the following steps as depicted in Figure 1.

- **Java** and **Java byte-code**. Either language can be used as input language to our model checker.
- **Jimple**. Generated by Java byte-code to Jimple translator, which we extracted from the Bandera framework.
- **SAL Level 1**. Generated by our Jimple to SAL translator.
- **SAL Level 0**. Generated by a trivial translator from SAL Level 1.
- **C++**. Generated from SAL Level 0 by our SAL to C++ translator. An executable of the model checker is obtained by compiling this C++ file together with the C++ model checking core.

# 3 Model Checking SAL

The SAL model checker does an exhaustive search of the state space by either a depth first or breadth first traversal. Starting from the initial state, it considers the set of rules whose guarding conditions are enabled in the state. For each enabled rule, the checker generates a successor state by executing the associated action. The resulting state is stored in a hash table so that it is not re-expanded if it is encountered again. A brute force approach would generate a new successor state for each enabled rule that is fired, where a rule would roughly correspond to one program statement. However, this results in many unnecessary interleavings between statements that are independent of one another.

As an initial step to combat state explosion, we adopt the strategy of executing a sequence of rules local to a thread atomically and interleaving threads only when a global operation is performed. In particular, the SAL model checker uses a technique called *atomic blocks* that Bruening developed for the Rivet Deterministic Tester [1]. Secondly, we use hash compaction to contain the otherwise unmanageable memory usage necessitated by large state spaces and state vectors. These model checking techniques are described in the rest of this section.

## 3.1 Atomic Blocks

The SAL model checker uses a form of partial order reduction called *atomic blocks* [1], that is driven by synchronization constructs and for which static analysis is not required. The main idea is to execute one thread as long as possible before generating a new state, after which other threads may be scheduled. This is safe provided the operations of the other threads can not possibly interfere with the operations of the current thread being executed. The algorithm differs from the usual VeriSoft-like approach where a global state is defined by the execution of a "visible" operation, i.e. an operation on a shared variable. Because threads in Java share the same address space, most variable accesses are potentially visible operations and the interleaving of threads may become too fine grained.

The atomic block method assumes that accesses to shared variables are always protected by locks. Although our tool does not currently enforce this, it will be extended with techniques used in data race detection tools for multi-threaded programs. Race detection methods based on the *happens-before* relation check that conflicting memory accesses from different threads are ordered by synchronization events [13] [6]. Another example is Eraser [15], a dynamic race detection tool developed at DEC. Eraser keeps track of what locks each thread owns when it accesses a shared variable at run time. When the set of locks one thread holds is disjoint from the set of locks another thread owns when they access a common variable, Eraser issues a warning since the variable should have been protected by a common lock. Finally, static type-based analysis has also been successful in detecting race conditions in large Java programs [7].

Having stated the assumptions, a brief summary of Bruening's atomic block algorithm is given here. At a state $s$, we generate its successors by taking each enabled thread in turn and executing it until

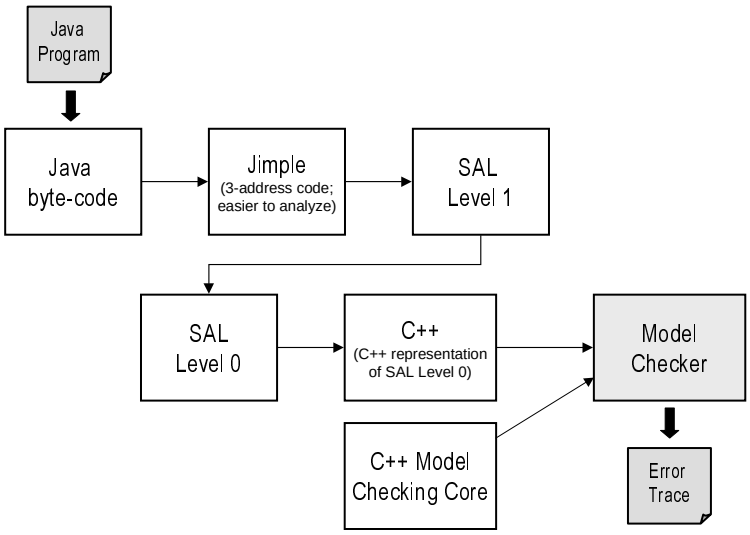



Figure 1: Translation steps in the model checker

an unlock operation is performed or the thread dies. Ignoring nested locking for the moment, note that in Figure 2(a) the sequence of statements (rules) between *Unlock(A)* and *Lock(B)* must only modify variables local to the thread since we assume that accesses to shared variables require the acquisition of locks. Hence, it is unnecessary to interleave such statements with statements from other threads.

Similarly, the statements within the synchronized region delimited by *Lock(B)* and *Unlock(B)* can be executed atomically since no other thread can access B while the lock is held. Furthermore, region X and region Y can actually be executed in the same atomic block because any operation by any other thread that may execute between the two regions must be independent of region X. We only have to consider schedules where such operations occur before region X. Therefore, we can safely execute each thread until an unlock is performed, at which point we return the new state.

In the case of nested synchronization blocks, the atomic block algorithm will span multiple locking operations. In Figure 2(b), for instance, the atomic block for T1 will begin with *Lock(A)* and end with the first unlock, namely, *Unlock(B)*. This may seem as though we are neglecting certain schedules, e.g., the case in which thread T2 modifies B while T1 has locked A but not B. Yet, such schedules *can* be ignored since we will consider the schedule in which T2 modifies B before T1 locks A; in terms of assertion checking, it does not matter whether T2 accesses B before or after T1 accesses A since A and B are independent.

## 3.2 Deadlock Detection

In general, a deadlock will be recognized by the model checker when a state has no successors and it is not a valid termination state. The only problem that atomic blocks pose is that certain deadlocks can be missed. In the case illustrated in Figure 2(b), the particular schedule required to realize the deadlock is never executed. As noted before, the atomic block algorithm will never execute the schedule in which T2 locks B immediately after T1 locks A but before T1 locks B. The lock-cycle deadlock that results from this schedule would have been caught if atomic blocks were delineated by locking as well as unlocking.

However, since larger atomic blocks result in fewer unnecessary interleavings, and in turn fewer states, we adopt Bruening's approach in delineat-

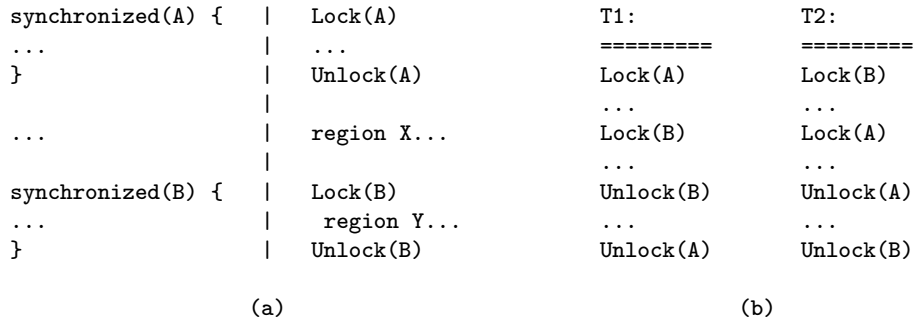



Figure 2: (a) Synchronized regions and atomic blocks. (b) An example of nested locking.

ing atomic blocks by unlocks only and detecting lock-cycle deadlocks by analyzing states for the existence of a cycle in the hold-wait relationship between threads. By keeping track of the most recently released lock for each thread, we can look for cycles where T1's most recently released lock is owned by T2, whose most recently released lock is owned by T3, and so on until we get to some thread Tn whose most recently released lock is owned by T1.

This check is linear on the number of threads, and is done only when a thread fails to obtain a lock (i.e. potentially closes a hold-wait chain into a cycle). Hence, the deadlock check introduces only a marginal overhead in most cases. Furthermore, a particular lock-cycle deadlock will always be detected (unless other deadlocks are detected first) since all schedules of atomic blocks are considered and the atomic blocks stop at each unlock, effectively considering all the locks each thread has accessed.

### 3.3 Implementation Specific Optimizations

We have also implemented simple optimizations that turn out to be critical in saving time and memory in model checking with atomic blocks. First, if an atomic block consists of more than one rule, a new state is instantiated once and each subsequent rule in the atomic block updates values in the same instantiation. This is permissible because intermediate states within the atomic block do not have to be stored during model checking. Since state vectors modeling dynamic structures like the thread stack and heap can get very large, creating a copy of the current state after each rule turned out to be a bottleneck in execution time. In some cases, the optimization led to an order of magnitude improvement in execution time.

Another source of run time overhead was the linear traversal over the set of rules required in determining which rules are enabled at each new state. Since most of the rules map to individual statements in a sequential process, the value of the program counter in the current state is used to locate in constant time the next enabled rule hashed on the PC value.

Finally, atomic blocks are aborted when a thread fails to acquire a lock and a successor state is not generated. This significantly reduces the number of schedules that need to be considered, reducing the size of the explored state space while preserving deadlocks and assertion failures. The informal justification is as follows. Suppose we have a thread T1 holding lock A and a thread T2 that blocks on the lock. The schedule in which T2 blocks on lock A does not have to be considered since the model checker will execute some schedule in which T1 first releases lock A allowing T2 to obtain the lock. Before T1 releases lock A, moreover, T1 can not access the same shared variables that T2 accesses before T2 tries to lock A since this will lead to a lock-cycle deadlock which our model checker will detect. (Note that locks can not be unlocked in the middle of an atomic block so T2 will be holding all of the locks on variables it accesses before blocking on lock A.)

Therefore, even when T1 is executed first, releasing lock A, the variables that T2 accesses before trying to lock A will not have been modified by T1 and it would effectively be as if T1 was signalled and allowed to continue. In addition, aborting atomic blocks when a thread blocks on a lock can not cause any deadlocks to go undetected since our lock-cycle deadlock checking functionality detects all deadlocks as long as atomic blocks end at each unlock and thread termination.

### 3.4 Hash Compaction

To reduce the memory requirements of the model checker, we have implemented hash compaction [18, 16]. Hash compaction reduces the memory requirements of the state table, which stores all states reached during verification and is used to decide whether a newly reached state is new or has been visited previously. Instead of storing the full state descriptor in this table, hash compaction stores only a (hash) signature. The memory savings come at the price of a certain probability that the verifier incorrectly claims that an erroneous protocol is correct. This probability, however, becomes negligibly small when choosing the signature size appropriately. Typical signature sizes are between 16 and 45 bits, resulting in memory savings of often more than two orders of magnitude.

## 4 Translation Steps

### 4.1 Translating Jimple to SAL

The SAL language is used to model the Java (Jimple) program as a state transition system. SAL has state variables, nondeterministic inputs, an initialization function, and a transition function that is a collection of guarded commands (or rules).

The state of the Java program is modeled in SAL using the following state variables: (1) each thread contains a program counter (`PC`), stack (array of frames), and stack pointer (`CurrentFrame`), etc.; (2) each object contains a class id, the fields, a counter for locking, etc. To select the next thread to execute, we use a nondeterministic input (`TID`).

Each Jimple statement is translated into a guarded command. For example, the Jimple statement

```
i0 = 1
```

is translated into

```
(PC[TID] = label_0) -->
  next(Stack)[TID]
    [CurrentFrame].localVariables.i0 = 1;
  next(PC)[TID] = label_1;
```

where `(PC[TID] = label_0)` is the guard condition, followed by the SAL statements that can be executed if the condition is true.

The translator has to deal with all advanced features of Java like inheritance, overriding, overloading, dynamic method lookup, exception handling, etc. For detailed descriptions of the semantics of these features, refer to the Java Virtual Machine Specification [14]. We believe that most of these features are easier to deal with at the Jimple level than at the Java source code level.

Exception handling, for example, is implemented by four SAL rules and several SAL tables that are generated at compile time. The first rule is passed the location label of the statement that throws the exception, and uses the `method_table` to look up the identifier of the method in which the exception is thrown. The second rule searches the `exception_table` for this method for a `catch` clause that handles the exception. A `catch` clause handles an exception only if the class of its parameter is the class of the exception or a superclass of the class of the exception. The `subclass_table` is used for this subclass relationship test. The third and fourth rules deal with the case that during the search no `catch` clause is found that handles the exception. Typically, the third rule is executed, returning from the method and re-throwing the exception. If the first stack frame of the thread had been reached, however, a Java run-time error occurred and this is signalled to the user.

### 4.2 Translating SAL to C++

The SAL description of the program is translated into a C++ source file that is #included in the model checking code, and the result is then compiled into an executable that outputs a trace if any deadlock or assertion failure exists. The included source file contains:

1. A C++ class with various accessors for each distinct type in the SAL description. Unbounded arrays are used to model data structures such as the stack and heap, and are important in storing per-thread data given that the actual number of threads can not be determined statically. Unbounded arrays are translated into dynamically resizing vectors where a designated default value represents the infinite sequence of array elements to the right of the last non-default element in the array. When an index $n$ greater than the current maximum index $k$ is accessed, the vector is resized to size $n+1$, letting elements at index $k+1$ through $n-1$ be default values. Likewise, when the last non-default element in the vector is set back to a default value, the vector is resized to size $k+1$ where $k$ is the index position of the next right-most non-default element.

2. Functions simulating SAL guarded commands. The *guard* function takes in the current state and the current values for the non-deterministic inputs and returns the next enabled rule to execute. The *apply* function then executes the enabled rule and any subsequent rules that fall under the same atomic block, and returns the resulting successor state to the model checking routines. Each successor state is generated by systematically incrementing the non-deterministic input values which includes the thread ID to schedule next.

# 5 Results

Table 1 gives the examples on which we tried the model checker. The source code for the examples is available at

`http://verify.stanford.edu/uli/java/`

The results we obtained for the examples are given in Table 2. For each example, we ran the checker under three different modes:

1. Noatomic: Atomic blocks turned off, interleaving of threads fined grained as possible at the level of individual rules.

2. LockUnlock: Atomic blocks delineated not only by unlocking but by locking as well. Does not require special lock-cycle deadlock detection method described in Section 3.

3. Unlock: Usual atomic blocks delineated by unlocking only.

Note that we achieve larger state reductions for the larger examples. In some cases, the reduction achieved by atomic blocks is three orders of magnitude. There is, however, many areas still in the implementation of the model checker where the execution time can be optimized.

Before hash compaction, moreover, the KSU Pipe example could only enumerate 140,000 states before running out of memory in the *noatomic* mode. With hash compaction, both the KSU Pipe and the ReaderWriter examples were able to be completed in the *noatomic* mode.

# 6 Conclusion

Tools to verify software have become almost a necessity as both the complexity of software and the extent of its use in critical systems are continually increasing. Given the usually intractable state spaces of software programs, different kinds of abstractions, optimizations, and "tricks" must be employed in concert to tackle the verification.

The SAL framework provides the means to easily integrate new and existing verification techniques around a common intermediate representation. This paper gives preliminary results on applying this framework to model checking Java programs. Support for dynamic data structures in SAL has made translating Java down to SAL direct and straightforward. Atomic block reduction and hash compaction demonstrate promising results in dealing with large state spaces and state vectors. We plan to extend the SAL model checker with additional state space reduction techniques, including symmetry reduction and a partial order reduction method called *persistent sets* [8] that will eliminate unnecessary interleavings even at the atomic block level. Yet ultimately model checking optimizations alone will not be sufficient to curb the state explosion problem. The model checker must be supplemented with front-end slicing and abstraction tools as well as other static program analysis tools. The SAL framework has provided a good starting point

Table 1: Example programs

| example | description | primitives | # prim. |
|---|---|---|---|
| Bruening's SplitSync | two threads access shared variable | synchronized | 2 |
| CS193k ReaderWriter | two reader and two writer threads | synchronized, wait/notify | 4 |
| CS193k TurnDemo | two threads synchronize using a semaphore | synchronized, wait/notify | 6 |
| NASA's classic | two threads communicate using events (deadlocks) | synchronized, wait/notify | 4 |
| NASA's ksu_pipe | 2-stage pipeline | synchronized, wait/notify | 4 |

Table 2: Results on example programs

| example | algorithm | states | rules | time | state reduction |
|---|---|---|---|---|---|
| Bruening's SplitSync | noatomic | 1763 | 4090 | | 1.0 |
| | lockunlock | 57 | 77 | | 53.1 |
| | unlock | 37 | 43 | | 95.1 |
| CS193k ReaderWriter | noatomic | 261 838 | 1030 130 | 442s | 1.0 |
| | lockunlock | 848 | 2184 | 1.91s | 309 |
| | unlock | 528 | 1356 | 1.54s | 496 |
| CS193k TurnDemo | noatomic | 26 145 | 68 715 | 30.4s | 1.0 |
| | lockunlock | 385 | 617 | 2.57s | 67.9 |
| | unlock | 166 | 236 | 2.40s | 158 |
| NASA's classic | noatomic | 45 924 | 118 047 | 46.6s | 1.0 |
| | lockunlock | 322 | 554 | 2.38s | 143 |
| | unlock | 143 | 234 | 2.11s | 321 |
| NASA's ksu_pipe | noatomic | 3990 883 | 14 022 723 | 6401s | 1.0 |
| | lockunlock | 28 357 | 92 334 | 51.5s | 141 |
| | unlock | 4991 | 15 762 | 11.8s | 800 |

to research the ways that different analysis tools interface and interact with each other.

# References


[1] D. L. Bruening. Systematic testing of multi-threaded Java programs. Master's thesis, MIT, 1999.

[2] J. Corbett, M. B. Dwyer, J. Hatcliff, S. Laubach, C. S. Păsăreanu, Robby, and H. Zheng. Bandera: Extracting finite-state models from Java source code. In *22nd International Conference on Software Engineering*, 2000. To appear.

[3] C. Demartini, R. Iosif, and R. Sisto. A deadlock detection tool for concurrent Java programs. *Software - Practice and Experience*, 29(7):577–603, 1999.

[4] C. Demartini, R. Iosif, and R. Sisto. dSPIN: A dynamic extension of SPIN. In *6th International SPIN Workshop on Practical Aspects of Model Checking*, 1999.

[5] D. L. Dill. The Mur$\varphi$ verification system. In *Computer Aided Verification. 8th International Conference*, pages 390–3, 1996.

[6] A. Dinning and E. Schonberg. Detected access anomalies in programs with critical sections. In *ACM/ONR Workshop on Parallel and Distributed Debugging*, pages 85–96, 1991.

[7] C. Flanagan and S. N. Freund. Type-based race detection for Java. In *ACM Conference on Programming Language Design and Implementation*, 2000. To appear.

[8] P. Godefroid. Partial-order methods for the verification of concurrent systems - an approach to the state-explosion problem. In *Lecture Notes in Computer Science*, volume 1032, 1996.

[9] P. Godefroid. Model checking for programming languages using VeriSoft. In *24th ACM Symposium on Principles of Programming Languages*, pages 174–86, 1997.

[10] The SAL Group. The SAL Intermediate Language. Technical report, UC Berkeley, SRI, Stanford University, 1999.

[11] K. Havelund and T. Pressburger. Model checking Java programs using Java PathFinder. *International Journal on Software Tools for Technology Transfer*. To appear.

[12] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–94, 1997.

[13] L. Lamport. Time, clock, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), 1978.

[14] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley Publishing Company, 2nd edition, 1999.

[15] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, and T. E. Anderson. Eraser: A dynamic data race detector for multi-threaded programs. *ACM Transactions on Computer Systems*, 15(4):391–411, 1997.

[16] U. Stern. *Algorithmic Techniques in Verification by Explicit State Enumeration*. PhD thesis, Technical University of Munich, 1997.

[17] R. Vallee-Rai and L. J. Hendren. Jimple: Simplifying Java bytecode for analyses and transformations. Technical report, Sable Research Group, McGill University, 1998.

[18] P. Wolper and D. Leroy. Reliable hashing without collision detection. In *Computer Aided Verification. 5th International Conference*, pages 59–70, 1993.

# On the Specification and Semantics of Source Level Properties in Java

**Radu Iosif**
Dipartimento di Automatica,
Politecnico di Torino
corso Duca degli Abruzzi 24,
10129 Torino, Italia
+39 011 564 7085
iosif@athena.polito.it

**Riccardo Sisto**
Dipartimento di Automatica,
Politecnico di Torino
corso Duca degli Abruzzi 24,
10129 Torino, Italia
+39 011 564 7073
sisto@polito.it

**Abstract**
Several efforts have been made recently towards practical source code model checking. This paper deals with a related problem that up to now has not yet been solved completely, namely the integration of LTL property specifications into object oriented source code. We present a notation that can be used for this purpose, and a sketch of the way a formal semantics can be assigned to such notation. We also address the problem of incorporating the specification pattern concept in our notation.

**Keywords**
Finite-state verification, temporal logic, Java.

## 1 Introduction

Thanks to the recent advances in tool support, model checking can now be applied with interesting results to the verification of non-trivial software systems. However, a number of problems still make the integration of model checking into the software development process quite slow. A first reason is that direct translation of programs written in high level languages into formal models usually yields excessively large models that cannot be approached with model checking. Some work has already been done in order to alleviate the above mentioned problem using various reduction techniques such as program slicing and abstraction-based specialization. Initially, the main focus was on the analysis of Ada programs, whereas more recently Java programs are being considered as well [1, 2, 6].

A second important problem that still remains is the fact that the specification of software requirements is not yet well integrated in common environments programmers are used to. More precisely, problems arise due the fact that the languages used for writing programs differ from those used for requirements specification, the latter being far more abstract and difficult to understand.

This paper addresses the latter problem specifically and outlines a possible solution based on a notation to specify properties within Java source code, in respect to the above considerations, i.e. a notation that can be well integrated in conventional object oriented software development. Such properties can then be verified using model checking. Our research focuses on Java programs, but the methodology could be extended to a large class of object oriented languages.

The paper is organized as follows: Section 2 describes the notation we use to write properties, Section 3 explains their semantics and Section 4 deals with specification patterns. Finally, Section 5 draws some conclusions and announces future work. The reader is assumed to have a basic familiarity with Java.

## 2 Specification Notation

Before illustrating the notation for specifying properties, it is worth making some preliminary considerations.

First of all, it is important to distinguish between source code and binary code verification. The former involves verification of properties associated with the program source code without requiring the compilation of the program, while the latter regards verification of already compiled code. In this paper we deal only with source code verification because it can be integrated into the software development process more easily and facilitates the programmer in specifying correctness requirements, since such requirements can be associated directly with the high-level program elements manipulated by the programmer, such as packages, classes and methods.

Another issue to be considered is that in many instances it is useful to verify properties associated with single classes or packages independently of the way they will actually be used. This is especially the case when distributed systems are considered, because in such systems server objects are made available to unknown clients which will eventually be developed later on. The verification of servers may be done having only an abstract specification of the client behavior. The need of verifying partially specified systems has already been recognized by other researchers, as reported in [5]. Here this need is addressed considering both properties related to the whole program and properties related to single classes of objects. For what concerns the latter, we also take into consideration the fact that the develop-

ment of object oriented (OO) software distinguishes between two programming roles: the developer role, which consists of implementing new classes that will be made available to other parties, and the user role, which consists of using already existing classes, without caring about how they are implemented. Consequently, it can be useful to divide the properties that can be expressed about a single class into two distinct subsets, according to the point of view under which they are formulated:

- interface properties, which are requirements that can be expressed by the class user and involve only interface elements (method invocations).
- implementation properties, which are requirements that can be expressed by the class developer and may also involve implementation details (class attributes).

A final preliminary discussion point regards the difficulty generally found in formalizing software properties expressed in a temporal logic language. To alleviate this problem, an important step ahead has been made introducing the concept of specification pattern [4] which is a mapping between properties expressed in natural language and temporal logic formulae. However, even if natural language properties are easy to understand, difficulties occur when they have to be applied in the context of a source code program.

Our notation has been defined taking this problem into consideration. We adopt the specification pattern concept and, in section 4 we discuss a possible way to integrate specification patterns into our notation.

As already explained, we have chosen to insert LTL formulae directly into the source code of the program as special annotation comments. Our choice was driven also by the need to use the same source file in both compilation and verification, without further modifications. In this way, properties become integral part of the source code of packages and classes, and are naturally carried with it. Each one of the following annotation forms can be used to specify a property:

```
/*@ property_specification */
//@ property_specification
```

A property has the following formal syntax:

```
property_specification ::=
  PROPERTY NAME = [quantifier] ltl_formula

quantifier ::=
  (FORALL|EXISTS) declarations [(predicate)]
```

The upper-case symbols denote terminals, while the lower-case ones are non-terminal symbols. The symbols enclosed in square braces are optional. The `declarations` symbol introduces a number of variables that range over the quantification domain, taking the classical Java form of a variable declaration list. The `predicate` symbol denotes a Java boolean expression used to restrict the quantification domain. An `ltl_formula` is obtained from any number of atomic propositions connected with the standard LTL operators. These include the temporal operators (`<>`, `[]`, `U`, `o`), the standard boolean operators and the implication operator `->`). The LTL composition rules apply here, such that, if `P` and `Q` are formulae, then `<>P`, `[]P` and `P U Q` etc. are also formulae.

Generally, we consider that properties annotated within the source code of a program are subordinated to the same scoping rules as any other language construct. That is, a property can only refer to the entities that are visible in its most enclosing scope. At the moment we are taking into consideration two kinds of scope: the package and the class scope. The package scope contains all classes and interfaces declared in the current package and the ones declared public in other packages. The class scope contains all fields, methods and inner classes declared in the current class or interface, along with the ones declared in other classes and packages that are visible in the current class. Properties declared inside a package scope but outside any class scope are denoted in what follows as package properties. Properties declared inside a class scope but outside any method declaration are denoted as class properties.

The interpretation of package and class properties is quite different, because a package property is an LTL formula related to the evolution of the global program state, whereas a class property applies to the evolution of the state of each instance of the class it refers to.

Let us now define the atomic propositions that can be used to build the various kinds of LTL properties. For package properties, atomic propositions take the form of boolean expressions involving static attributes of the package classes. For class properties instead, we consider separately interface and implementation properties.

The specification of interface properties makes use of the following two atomic propositions related to the occurrence of method call and return events:

- `calling(m [, argument_list])`
- `returns(m [, argument_list] [, x])`

where square brackets indicate optional fields.

Predicate `calling(m [, argument_list])` is true in all object states where some call to method m with actual arguments `argument_list` is being executed. If method m does not have arguments, the argument list is void. In practice, this atomic proposition becomes true in the evolution of a class instance whenever a call to method m with actual arguments `argument_list` is issued on that instance, and it remains true until the corresponding method execution terminates. Of course, in a concurrent environment it is possible to have time-overlapping executions of m, in which case the predicate remains true until the current number of concurrent executions of m with actual arguments `argument_list` reduces to 0.

Predicate `returns(m [, argument_list] [, x])` is true in a certain object state provided that the last interface event occurred in the object is the return of value x from a call to method m with actual arguments `argument_list`. As with the previous predicate, the argument list and the return value can be missing (e.g., if m does not have arguments, and it does not return any value), `returns(m)` is true if the last event occurred in the object interface is a return from `m`.

The specification of implementation properties can use the same atomic propositions defined for interface properties, as well as additional properties related to the class implementation. Generally, every boolean expression which is legal in Java is also an atomic proposition. The evaluation of an atomic proposition must complete normally and cannot have side effects.

In addition to boolean expressions, three special atomic propositions are introduced, in order to represent information about the program control flow [9]:

- `at(l)` is defined for a program label l; this proposition is true in a certain program state if there is an active thread whose current control location is l.
- `nextT(l)` is defined for program label l; this proposition is true in a certain program state if the next state is obtained by executing the program statement labeled by l.
- `nextP(t)` is defined for a program variable t which is a reference to a thread object; this proposition is true in some state if a program statement executed by the thread referred to by t leads to the next state.

An example of a syntactically valid interface property specification is:

```
class C {
  /*@
    property neverGetNegative =
    forall int x (x < 0) [](!returns(get, x))
  */
  int get() { ... }
}
```

The specification of class properties is inherently object oriented. They are automatically inherited via subtyping, as a consequence of the fact that instances of a subclass are also instances of its superclasses, therefore all properties that apply to a superclass should also apply to its subclasses.

## 3 Property Semantics and Validation

Writing temporal logic properties in terms of source program expressions in the notation introduced in section 2 has an intuitive meaning related to the common understanding of language constructs. Nevertheless, as these properties must be verified formally, their semantics must be expressed in a formal way.

The formal understanding of the property language introduced in Section 2 is tightly related to the execution semantics of a Java program. We have already developed a detailed behavioral model of Java programs based on labeled transition systems (LTS), which is presented in [7]. Automatic translation of the Java source code into this model is currently being implemented.

Here we focus on class properties because the interpretation of package properties is straightforward and quite standard. Since class properties are related to the evolution of class instances, their interpretation can be defined with respect to a LTS model representing a class instance evolution. It is worth noting that the level of abstraction needed for such LTS model depends on the kind of class property considered. The abstraction level needed for interface properties is higher than the one needed for implementation properties, which is an obvious consequence of the way these properties were defined in Section 2. An interface property involves only method call and return events, while an implementation property refers also to the state of object fields, static fields and synchronization monitors implicitly associated with objects.

According to the above considerations, we have defined two distinct LTS models, one for defining the semantics of implementation properties and the other one for interface properties. Let us call them LTSm and LTSn, respectively. In LTSm, a state is explicitly composed of all information regarding the individual state of the object, including its associated synchronization monitor. A transition in LTSm is fired by the execution of a program statement. In LTSn instead, a state is represented as the ordered set of method call and return events that

have occurred in the past in the object interface. A transition of LTSn is fired by a method call/return event. In practice, LTSm is a refinement of LTSn, in the sense that each state in LTSn (interface state) corresponds to a sequence of states in LTSm (implementation states). For example, an interface state in which atomic proposition `calling(m)` is true corresponds to a sequence of implementation states representing intermediate steps in the execution of m. A complete formalization of the interface and implementation LTS models is presented in [8]. It is easy to see that, as interface and implementation properties can be interpreted on LTS models, they can also be translated into a model checker input language (e.g., PROMELA never-claims).

This kind of relationship between LTSm and LTSn makes it possible to interpret interface properties both on LTSn and on LTSm. Of course, it is important that the truth value of an interface property P is the same for an execution path of LTSn and for the corresponding path of LTSm, because in this case we can claim that P holds in LTSn iff it holds in LTSm and vice-versa. An LTL formula is closed under stuttering if its truth value is not affected by the addition/deletion of stuttering steps in the execution path on which it is evaluated, where a stuttering step is one which does not alter the truth value of the atomic propositions occurring in the formula. We claim that the implementation states corresponding to an interface state are stuttering steps with respect to interface properties. As a consequence, the interpretation of interface formulae that are closed under stuttering is the same in LTSm and in LTSn. A formal proof is given in [8].

This claim is useful in order to validate interface properties using model checking. Informally, we consider that an interface property is validated if it holds in every implementation of the interface. For example, in the following Java code:

```
import java.util.Vector;
import java.util.LinkedList;

interface Stack {
  void push(int info);
  int pop();

  /*@
    property Consistency =
      forall int x, y, z (x != y)
      (returns(push,x) U (!returns(push,y)
       U returns(pop,z))) -> x == z
  */
}

class VectorStack implements Stack {
  Vector data = new Vector();
  int top;
  public synchronized void push(int info) {
    data.add(top ++, new Integer(info));
  }
  public synchronized int pop() {
    Object info = data.remove(-- top);
    return ((Integer) info).intValue();
  }
}

class ListStack implements Stack {
  LinkedList data = new LinkedList();
  public void push(int info) {
    data.addFirst(new Integer(info));
  }
  public int pop() {
    Object info = data.getFirst();
    return ((Integer) info).intValue();
  }
}
```

the interface property `Consistency` informally says that if a `push(x)` is followed by a `pop()` with no intermediate other `push(y)`, then the return value of `pop()` is `x`. The property is automatically inherited by the `VectorStack` and `ListStack` classes. Its validation reduces to the validation of both implementation properties in the two implementing classes. It can be easily seen that the `VectorStack` implementation respects the property because both `push` and `pop` are synchronized, preventing for concurrent thread access to the stack data. On the contrary, the `ListStack` implementation violates the property in concurrent access. Let us consider two separate threads, `t1` and `t2`, each one performing pushes and pops on the same instance of the class `ListStack`. If `t1` has pushed the value `x` and its call to `push` returns and if, afterwards, `t2` pushes the value `y` but, before its call to `push` returns, `t1` issues a call to `pop` which returns before the second call to `push`, then the returned valued is `y`, instead of `x`.

If in the future other classes will be defined by specialization from the Stack interface, they will also inherit the Stack interface properties, which will have to be verified.

As already mentioned, the validation of an implementation property P specified within a class reduces to the validation of P in every class instance. We call the latter an instance property. The truth of an instance property is defined on execution sequences in LTSm starting with the state in which the instance is actually created. Although in Java objects may be garbage collected, we consider for simplicity that their life does never stop (when they are no longer referenced, their state can be

assumed to remain unchanged forever).

To conclude, the validation of interface and implementation properties follows a top-down model. An interface property holds if every corresponding implementation property holds, while an implementation property must hold for every instantiation of the class. As mentioned before, using existent model checker tools (e.g., SPIN) to verify program properties is possible because the model checkers input languages (e.g., PROMELA) essentially describe labeled transition systems.

In practice, we approach the verification problem bottom-up. As model checking deals with actual program states, it can be used to check instance properties, one at the time. When all instances of a class are proven to satisfy a property, we say that a class property is satisfied. Furthermore, if all implementing classes are proven to satisfy an interface property, then the interface property holds.

Using model checking, one can decide if an LTL formula holds for a system, given that the system state space is finite. For the moment, we formally proved that the finiteness of the program state space is also a sufficient condition for the decidability of class properties quantified over finite domains. However, more work has to be done in order to make the decision procedure cost effective. Even if finite, quantification domains can still be large enough to make the decision very expensive in time and space. This problem can be overcome using program abstraction techniques [1], and is considered as further work.

An important problem faced by existent verification tools is the lack of underlying support for dynamic memory management and polymorphism (i.e., dynamic method dispatch). In order to alleviate this problem, we have extended the model checker SPIN, providing efficient embedded support for the modeling of dynamic run-time information [3].

## 4 Specification Patterns

The specification of behavior properties expressed in LTL is generally considered a difficult task. Even simple properties can be erroneously formalized, which may lead to spurious error reports, while real errors could be neglected. As mentioned in Section 2, the concept of specification pattern was introduced in order to facilitate properties specification. In this section we explore the possibility of encapsulating specification patterns into Java interfaces and applying them to an existing source code program by means of inheritance.

Let us first note that a specification pattern is always parameterized with respect to a number of atomic propositions or events. In our state-based model, any observable state change is considered to be an event. The meaning of "observable" is however application dependent. Introducing a general observability criterion may greatly increase the size of our model, therefore we chose to leave this task to the implementor by introducing the concept of probe methods. A method M is a probe method if:

- during its execution the state of the object does not change in an observable way, from the user point of view;
- the execution of M does not block the calling thread, not even temporarily, nor it completes abruptly by throwing an exception.

Ideally, the execution of a probe method is performed without interleaving with other threads. In practice, a probe method is a way to get some state information out of an object without interfering with its behavior. A probe method which returns a boolean value may be used to trigger an event. A specification pattern can be introduced by means of an interface which declares a number of probe methods. For example:

```
interface Response {
  boolean P();
  boolean S();

  //@ property Response = [](P() -> <> S())
}
```

introduces the pattern of response. The pattern's informal intent is to describe the cause-effect relationship between event P() and event S(). In order to apply this pattern to an already existing class, the implementor must first make it inherit from the pattern interface and consequently, implement the P() and S() methods according to the actual events. Let us consider the following class implementing a dialog box with an edit field. The actual requirement is that every read operation must be followed by a write operation.

```
class EditDialog implements Response {
  protected EditField editField;
  protected TextField textField;

  String info;

  void readText()
  { info = editField.getText();
    p = true; p = false;
  }

  void writeText()
  { s = true; s = false;
```

```
    textField.putText(info);
  }

  // P() probe method implementation
  private boolean p = false;
  boolean P() { return p; }

  // S() probe method implementation
  private boolean s = false;
  boolean S() { return s; }
}
```

In this example the probe methods are P() and S(). In order to trigger the method invocation events, we use the boolean variables p and s. In this particular case, the observable state of an EditDialog object from the response pattern's point of view is given only by the pair of boolean values (p, s).

## 5 Conclusions and Future Work

The notation introduced in this paper to specify properties associated with Java classes and packages contributes to the elimination of the gap between formal specification languages and programming languages, without sacrificing the accuracy of formal reasoning. In particular, the notation is integrated into the object oriented paradigm, which makes it easy to understand by programmers.

We plan to continue the work presented here in various ways. First, we intend to define the meaning of properties expressed in terms of method local variables. Another further direction regards the possibility of combining interface property patterns in order to create a flexible specification system.

Finally, computational support has to be provided for our notation that is, a software tool able to automatically check properties annotated within the source code of the program.

## REFERENCES


[1] J. Corbett, M. B. Dwyer et al., "Bandera: Extracting Finite-state Models from Java Source Code", To appear in Proc. 22nd International Conference on Software Engineering, (June 2000).

[2] C. Demartini, R. Iosif, and R. Sisto, "A deadlock detection tool for concurrent Java programs", Software: Practice & Experience, Volume 29, Issue 7, (July 1999) 577–603.

[3] C. Demartini, R. Iosif, and R. Sisto, "dSPIN: A Dynamic Extension of SPIN", Lecture Notes in Computer Science, Vol. 1680, pp. 261–276 (September 1999).

[4] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in Property Specifications for Finite-state Verification", Proc. of 21st International Conference on Software Engineering, ACM Press, May 1999, pp. 411-421.

[5] M. B. Dwyer, and C. S. Pasareanu, "Filter-based Model Checking of Partial Systems", Software Engineering Notes, Vol. 23, 6, pp. 189-202, ACM Press, November 3-5 1998.

[6] K. Havelund and T. Pressburger, "Model Checking Java Programs Using Java PathFinder", To appear in the International Journal on Software Tools for Technology Transfer.

[7] R. Iosif and R. Sisto, "A Formal Execution Model for Java Programs", DAI-ARC Technical Report, Politecnico di Torino, 2000.

[8] R. Iosif and R. Sisto, "Temporal Logic Properties of Objects", Submitted for presentation at the Fourth International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS) 2000.

[9] A. Thayse et al., *From Modal Logic to Deductive Databases: Introducing a Logic Based Approach to Artificial Intelligence*, Chapter 4, Wiley, 1989.

# Toward Synergy of Finite State Verification and Testing

Gleb Naumovich
Phyllis G. Frankl *
Polytechnic University
6 Metrotech Center
Brooklyn, NY 11201

{gleb, pfrankl}@duke.poly.edu

## Abstract

Finite state verification (FSV) and testing are usually viewed as competing approaches to software validation. In this short paper, we propose a technique for combining FSV synergistically with testing, with the goal of identifying faults more quickly and with less manual effort than with FSV alone and more effectively than with testing alone. We propose using information about potential faults obtained during the FSV analysis to direct selection, execution, and checking of test data, with the intent of confirming these faults.

## 1 Introduction

In finite state verification, a finite model of the system is constructed, usually abstracting away many details, and the FSV tool (verifier) explores the state space to determine whether a given property $P$ holds. The model is constructed in such a way that if the verifier determines that $P$ holds for the model, then $P$ also holds for all possible executions (and hence, for all possible test data) of the actual system. In this case, there is no need to test the system for the behaviors captured by $P$. On the other hand, if the verifier finds a violation of $P$, it may or may not reflect a property violation in the actual system. Such violation is *spurious* if no violation-revealing path through the system model corresponds to a feasible execution of the system. Normally, given a representation of the property violation on the model, the human analyst (or simply *analyst* hereafter) has to decide whether this violation appears spurious, in which case the analyst has to refine the system model, providing more detail, and then re-run the verifier. The additional details may allow the verifier to determine that $P$ is always satisfied or there may still be a violation. In the latter case, the process continues.

This incremental approach to FSV has several weaknesses related to the presence of human factor in the verification process. First, this approach relies on the analyst to decide whether a property violation found by the verifier is spurious or not, which is time-consuming and error-prone. Second, the analyst can only review one property violation at a time, while our technique can use information about all found violations at the same time. Finally, if a violation appears feasible, it is important to analyze a real execution of the system that results in this violation, so that the error in the system can be found and removed. Unfortunately, debugging cannot be used until the analyst manually identifies test data that are likely to produce such an execution.

Our proposed technique uses testing, along with model refinement, to address these weaknesses. When a property violation is found by the verifier, the following steps are performed in parallel:

- An automated testing tool uses information developed during the FSV analysis to direct selection, execution, and checking of test data, with the hope of finding data that shows the violation to be real, and
- The analyst refines the model and re-starts the verifier. Note that the analyst only needs to pick a reasonably important aspect of the system to be modeled during the next run of the verifier, without having to worry about whether the violation is spurious.

If the violation is real, testing will sometimes be able to find test data that exhibits this violation in the system. In this case, the parallel FSV session can be stopped and a debugging session with the found test data can be started. If the violation is spurious, thorough testing of relevant parts of the system may help increase confidence that such is the case, but, of course, will never be able to prove it. The gain in this case is that the analyst is able to start a new, more precise, verification session promptly, which helps to speed up the overall process of FSV. Thus, from the point of view of the FSV analyst, this approach saves some manual work; from the point of view of the tester, this approach helps direct testing effort toward execution paths that are at risk of violating the specification.

In the remainder of this paper, we refine these ideas further, illustrating with a simple example. Section 2 summarizes relevant background on FSV and testing, Section 3 illustrates the technique and discusses some of the issues, and Section 4 concludes.

*Supported in part by NSF Grant CCR-9870270.

# 2 Background

## 2.1 Background on finite state verification

Conceptually, many FSV approaches represent the system under analysis as a collection of states in which this system can be during its executions and transitions connecting these states. This construct may be created explicitly (e.g. [4,6,12]) or implicitly (e.g. [9,15]). For simplicity, in this paper we use an explicit representation of the state space, although the proposed techniques can be extended for implicit representations.

Consider the example in Figure 1. The two threads of control, `T1` and `T2`, that comprise this system are represented as FSAs in Figure 1(a). State 0 is the start state in both FSAs. The states representing termination of the threads are indicated with double circles. The transitions between the states are labeled with events in the threads to which they correspond. For example, the transition from state 1 to state 2 of thread `T1`, labeled `start T2`, represents thread `T1` starting thread `T2`. Events `a` and `b` represent some events in the threads that are relevant to the property. Square brackets that follow some events represent conditions on when the event is executed. For example, state 2 of thread `T1` represents this thread before executing an `if` statement with predicate $\texttt{x} > 0$. Event `a` appears on the branch of this `if` statement that is executed when the predicate evaluates to true. $\tau$ denotes an empty event, representing absence of any events. For example, the $\tau$-based transition from state 2 to state 3 of thread `T1` means that nothing of interest happens on the branch of the `if` statement that is executed when $\texttt{x} \leq 0$. Note that we assume that `x` is a shared variable that is an input to thread `T1` and is not changed by either `T1` or `T2`.

Formally, we can represent an *FSA* as a tuple $(S, s_0, \Sigma, T)$, where $S$ is the set of states, $s_0$ is a unique *start* state, $\Sigma$ is the set of events, and $T$ is the set of *transitions*. We use the notation $s_1 \xrightarrow{e} s_2$ to represent a transition based on event $e \in \Sigma$ from state $s_1 \in S$ to state $s_2 \in S$. A *path* through an FSA on an event sequence $e_1, ..., e_n$ from $\Sigma$ is a sequence of transitions $s_0 \xrightarrow{e_1} s_1 \xrightarrow{e_2} ... \xrightarrow{e_n} s_n$.

In this paper, we assume that FSA-based models of the threads of control are derived from the source code for the system. While construction of models based on high-level descriptions is attractive and has been advocated for FSV [11], since testing is used in our approach, we need a direct mapping between the thread models and the executable code for the system.

A *property* about a software system is a representation of either desirable or undesirable behavior of this system. We define properties in terms of the events observed in the system, using FSAs with a special *violation* state $v$. The violation state is a sink: $\forall e \in \Sigma_P, v \xrightarrow{e} v$. A property is *violated* on an execution that corresponds to the event sequence $p = e_0, e_1, ...$, if the path through the property FSA on this sequence ends in the violation state.

Figure 1(b) shows a property specifying that on no execution of the system can event `b` be observed if by that time event `a` has been observed an odd number of times. For example, the sequence of events `a`, `a`, `a`, `b` corresponds to the path $0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{a}} 0 \xrightarrow{\texttt{a}} 1 \xrightarrow{\texttt{b}} v$, and so violates this property. Note that events other than `a` and `b` do not affect this property, which means that if, for example, event `start T2` is contained in a sequence of events, it does not change the current state of the property.

A *reachability graph* represents all reachable states of the system, to the extent that this system is modeled by the FSV technique of choice. In our example, each thread of control in the system is modeled with an FSA, so a state of the system can be represented as an ordered collection of FSA states, one for each thread. The reachability graph is a cross-product of the FSAs for all threads. Figure 1(c) contains the reachability graph for our example.

Paths through the reachability graph represent executions of the system. A path in the reachability graph is *executable* if it corresponds to a real execution of the system. All other paths are *spurious*. If there is a path from the start state of the reachability graph to some state $s$, such that the property is violated on this path, $s$ is called a *violation* state.

Many FSV approaches are capable of checking two general kinds of properties, safety and liveness. Safety properties are always finitely refutable and liveness properties are never finitely refutable [1]. The approach proposed in this paper deals only with safety properties, since the infinite nature of liveness properties means that an execution that represents a violation of a liveness property is infinite and thus cannot be reasoned about using testing techniques[1].

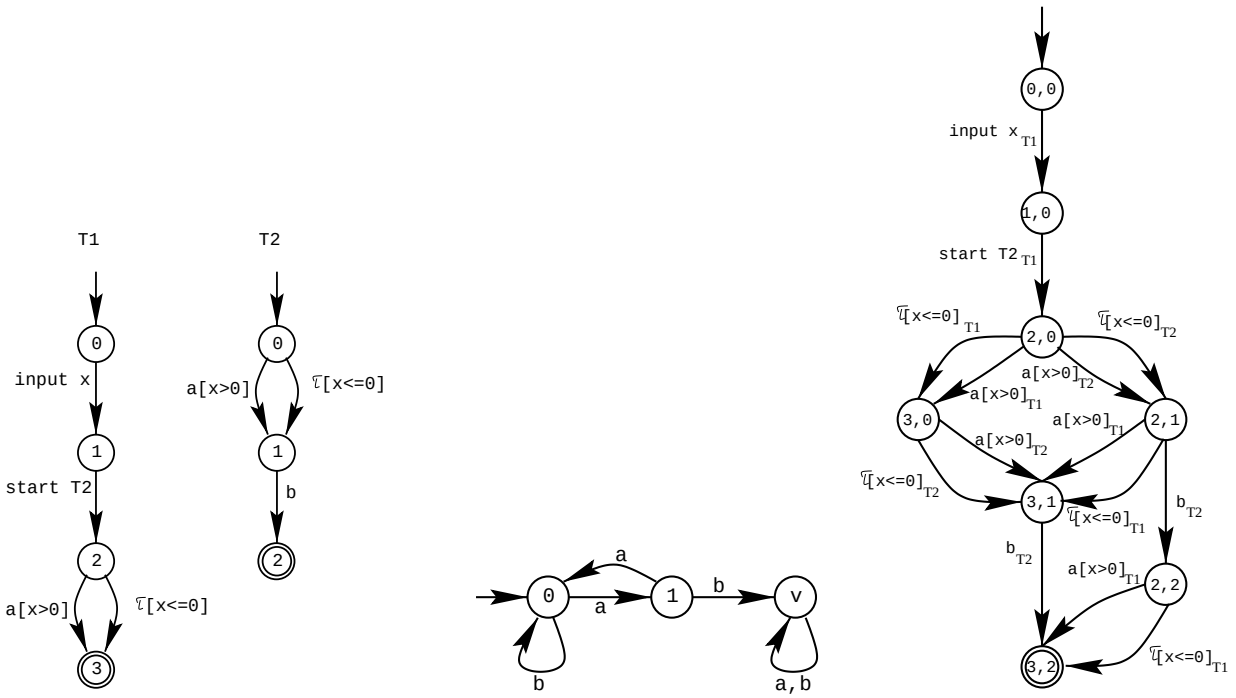
The goal of our approach is to combine FSV and testing to either prove, with respect to a given property, that no violation states exist in the reachability graph or to find an executable path from the start state to a violation state of the reachability graph.

## 2.2 Background on Testing

Whereas FSV primarily aims to prove that the specification is satisfied, testing aims to find faults, i.e., to demonstrate that the specification is not satisfied. To test a piece of software, one selects test cases from the input domain, executes the software on each test case, and checks whether the results satisfy the specification. In addition, one might monitor which path through the program is executed by each test case (or other aspects of the execution that are not immediately observable) or might attempt to force execution of a particular path.

Many testing techniques involve analyzing the control flow (and/or data flow) of the program then requiring the test data to execute representatives of certain

[1] In addition to safety and liveness properties, there are properties that are neither safety nor liveness, but any such property can be represented as a conjunction of a safety property and a liveness property [2]




(a) FSAs for the threads of control (b) Property (c) Reachability graph

Figure 1: A reachability graph-based example

classes of program paths. These techniques were originally developed for testing sequential programs but can be extended to testing concurrent programs [13,18,19]. Testing criteria of this nature often result in a large number of test requirements, even for moderate-sized sequential programs. For concurrent programs, the explosion in the size of the state space makes this problem even more severe. Thus, the tester needs guidelines for selecting portions of the state space that should be explored. In the proposed technique, those guidelines are supplied through interaction with FSV.

One of the most difficult aspects of testing is the *oracle* problem, i.e., the problem of determining whether the result of a particular test case satisfies the specification. The use of formal specifications can significantly alleviate this problem, by allowing test results to be checked automatically [17]. In particular, techniques have been developed for automatically generating test oracles from specifications written in temporal logics, as are commonly used in FSV [7,8,16].

In testing and debugging concurrent programs, special problems arise due to non-determinism. A given test case may expose a fault on some executions, but not expose it on others, due to differences in the interleavings of statements from different processes. If executing test case $t$ does not expose a fault, it may be useful to re-execute it many times to check different interleavings. If executing test case $t$ does expose a fault, it may be difficult to reproduce the interleaving in order to debug the program. In order to deal with these issues, testing environments for concurrent programs have been proposed in which the interleaving of processes is monitored or controlled [3].

# 3 Using Property Violations to Guide Testing

When the finite state verifier finds a property violation, we would like to use this information to guide testing. There are two ways in which we would like to guide testing of concurrent systems, by choosing appropriate test data and by choosing scheduling of relative execution of the threads in cases where they can execute independently from each other. The former is a general problem of testing methods and the latter is specific to concurrent systems. In this section we describe several different approaches to using information produced by reachability analysis to guide testing-based search for property violations.

## 3.1 Choosing Thread Scheduling

We will assume that our testing approach has instrumentation that lets us at any point to force execution of the current instruction from any of the threads that are not blocked. (Such instrumentation can be either embedded in the run-time execution environment or done on the source code level, similar to [3].) We use information about the violation states in the reachability graph to force testing to exercise those thread interleavings that improve chances of finding a real execution. To

this end, we introduce the notion of *interleaving selection criterion ISC* as a predicate defined on the set of all transitions in the reachability graph. This criterion evaluates to true if the transition should be explored, if possible, during testing and false if the preceding run of the verifier does not indicate that taking this transition can lead to a violation state. During testing, we apply this criterion to all transitions that correspond to thread interleavings that can be taken from the current state. If multiple transitions may be taken, according to the criterion, the testing tool will pick one of them and thus drive execution of the test case. If no transition from the current state of the reachability graph can be found that satisfies the criterion, the testing tool will backtrack to an earlier point in the execution and pick an alternative interleaving.

There are two different forms in which verifiers can return information about property violations. One of them is a set of violation states in the reachability graph and the other is a set of paths to some violation states in the reachability graph. Suppose first that $V$ is the set of violation states returned by FSV. An intuitive criterion based on this set is

$$ISC_V(s_1 \xrightarrow{a} s_2) = \begin{cases} \text{false} & \text{if } \forall v \in V, v \text{ is not reachable} \\ & \text{from } s_2 \\ \text{true} & \text{otherwise.} \end{cases}$$

Consider Figure 1(c) and violation state $(2,2)$ found by the verifier used. Suppose that the value of `x` was randomly chosen to be 5 when executing code corresponding to the transition between states $(0,0)$ and $(1,0)$. Consider the point during program execution immediately after thread `T2` has been started by thread `T1`, which corresponds to state $(2,0)$ of the reachability graph. In this state, the two threads may be executed in parallel, and so different event interleavings are possible. One possibility is to execute the `if` statement of `T1`, which means event `a`, because of our choice of value of `x`. This corresponds to the transition to state $(3,0)$ of the reachability graph. Since the violation state $(2,2)$ is not reachable from $(3,0)$, this interleaving will not lead to the violation found by the FSV session, and so will not be taken during testing. The other possible interleaving at state $(2,0)$ is to execute the `if` statement in thread `T2`, which corresponds to the transition on `a` from $(2,0)$ to $(2,1)$. Similarly, out of two possible interleavings at state $(2,1)$, the testing run will choose executing the code corresponding to event `b` in `T2`. At this point, we have detected a violation of the property with testing.

Many verifiers are capable of returning a path or a set of paths to some violation states in the reachability graph. Suppose that $W$ is such a set of paths. An intuitive criterion based on this set is

$$ISC_W(s_1 \xrightarrow{a} s_2) = \begin{cases} \text{false} & \text{if } \forall w \in W, w'(s_1 \xrightarrow{a} s_2) \text{ is not a prefix of } w, \\ & \text{where } w' \text{ is a path traversed up to state } s_1 \\ \text{true} & \text{otherwise} \end{cases}$$

This criterion stipulates that a transition should not be explored if it cannot lead to execution of a path in $W$. Assume that the verifier returned a violation path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$. Consider a point of the program execution corresponding to state $(2,0)$ of the reachability graph. If the `if` statement of `T1` is executed at this point, this path will not be followed, and so the testing tool has to execute the `if` statement of `T2`.

Intuitively, $ISC_W$ is stronger than $ISC_V$ in the sense that following a violation path during testing (if it is feasible and test data are adequate) always leads to a violation of the property, while entering a violation state does not necessarily represent a violation, because the path taken to this violation state during testing may be different from any of the paths that represent the violation.

There may be situations in which $ISC_W$ is preferable and situations in which $ISC_V$ is preferable. State $(3,2)$ of the reachability graph is a violation state, since the graph contains the path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\tau\texttt{[x}\leq\texttt{0]}} (3,1) \xrightarrow{\texttt{b}} (3,2)$ that violates the property. (There is also another violation path to this violation state.) Suppose first that we use this violation path in the interleaving selection criterion. Since this path is spurious, no choice of test data will exercise it. Thus, on each test case, the testing tool will stop execution because the given path cannot be exercised, even though these test cases could potentially execute a different violation. Now suppose that we use violation state $(3,2)$ in the testing criterion. Even though none of the violation paths to this state are feasible, testing could still find a violation by examining a real violation path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$. Note that this path leads to a different violation state, $(2,2)$, but $ISC_V$ permits that, because state $(3,2)$ is reachable from $(2,2)$.

Alternatively, suppose that our testing criterion is based on the violation state $(2,2)$. Suppose that the value of `x` used in our test is negative. In this case, path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\tau\texttt{[x}\leq\texttt{0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$ can be taken. Even though the violation state is reached, testing did not find a violation of the property, because at the end of this path the property is in state 1, which is not a violation state. If, instead of a violation state, the verifier returns the path $(0,0) \xrightarrow{\texttt{input x}} (1,0) \xrightarrow{\texttt{start T2}} (2,0) \xrightarrow{\texttt{a[x>0]}} (2,1) \xrightarrow{\texttt{b}} (2,2)$, testing with a negative value of `x` will be stopped early, at state $(2,0)$, because transition $(2,0) \xrightarrow{\texttt{a[x>0]}} (2,1)$ cannot be taken.

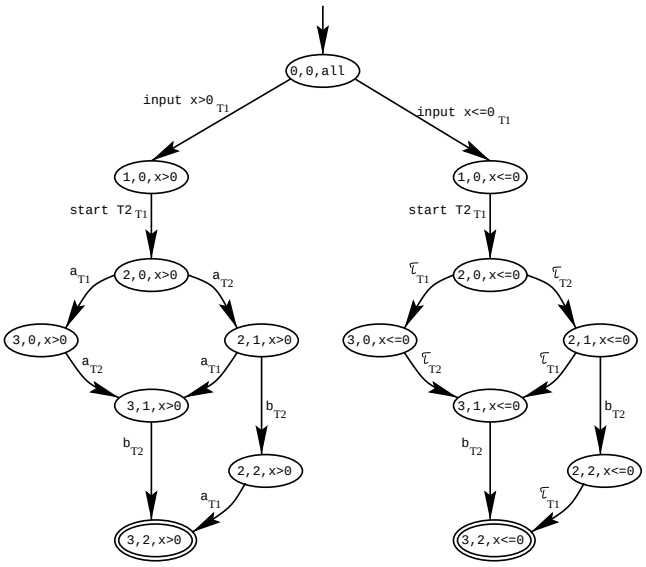



Figure 2: The reachability graph modeling variable `x`

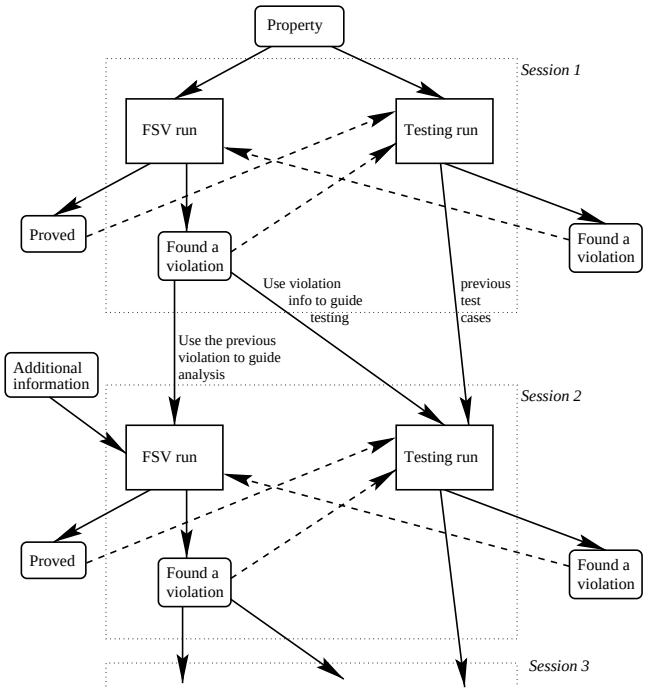



Figure 3: The proposed process of simultaneous use of FSV and testing

### 3.2 Choosing Test Data

Consider the problem of choosing appropriate test data. In general, choosing input data to follow a path through the reachability graph in such a way that it correlates with the values of modeled variables is undecidable. In practice, it may be possible to use symbolic execution [5] or some heuristics [10, 14] or to use random test data, aborting those executions that are not exploring the part of the reachability graph of that is of interest. In this section we propose an approach for choosing values of variables that are modeled by the verifier.

Many FSV approaches are capable of modeling system variables and including them in the analysis. In this case information about these variables (either the actual values or approximations, such as sets or intervals of values) can be used by testing to choose input data for this variable. For example, behaviors of variable `x` in our example in Figure 1 can be modeled by including the sign of `x` in the states of the reachability graph, as shown in Figure 2. Each state in this reachability graph is labeled $(s_1, s_2, r)$, where $s_1$ is the state of thread `T1` from Figure 1(a), $s_2$ is the state of thread `T2`, and $r$ is the range of values of variable `x`. In this example we consider only three possible ranges, $\mathtt{x} > 0$, $\mathtt{x} \leq 0$, and `all`, which denotes all possible values of `x` (the latter appears only in the start state of the reachability graph).

Suppose that we use violation state $(2, 2, \mathtt{x} > 0)$ found by FSV to drive testing. The simplest approach in this case is to use the range $\mathtt{x} > 0$ in test data selection, since we can perform an additional static analysis and detect that, once selected, the value of `x` does not change in this program.[2] This choice is even easier if path-based test criterion is used, because it is only the range of values of `x` that appears in the states along this path that has to be analyzed.

[2] If `x` is redefined, the problem can be much more difficult.

Modeling variables in this way in the reachability graph can help not only in selecting test data, but also in quickly discarding test data that are not likely to lead to violation. For example, if a negative value of `x` is chosen for testing, then after transition $(0, 0, \mathtt{all}) \xrightarrow{\mathtt{input\ x} \leq 0} (1, 0, \mathtt{x} \leq 0)$ in the reachability graph in Figure 2 is taken during testing, we can stop execution of the test, because none of the violation states in the reachability graph are reachable from state $(1, 0, \mathtt{x} \leq 0)$.

## 4 Conclusion

We have proposed a technique for using testing and finite state verification synergistically in the attempt to verify or disprove properties of concurrent systems. The process for applying this technique is illustrated in Figure 3. In our approach, testing and model refinement are both used to explore violations returned by the verifier, in order to determine whether those violations represent real executions or are spurious. Information returned by the verifier describing violation states and/or paths to those states is used to guide testing. Test cases are run in an environment where interleavings can be controlled and where executions can be aborted if it is determined that they will not be able to reach given violation states or execute a given paths to violation states.

This approach may offer a more efficient way to determine whether a violation is spurious than model refinement alone. In addition, finding test data that causes a property violation can be useful for identifying and removing the fault. We plan to implement our ap-

proach and to carry out experiments aimed at determining whether it is indeed useful.

## References


[1] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems*, 15(1):73–132, Jan. 1993.

[2] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2:117–126, 1987.

[3] R. Carver and K.-C. Tai. Replay and testing for concurrent programs. *IEEE Software*, pages 66–74, Mar. 1991.

[4] S. C. Cheung and J. Kramer. Compositional reachability analysis of finite-state distributed systems with user-specified constraints. In *Proceedings of the 3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 140–151, Oct. 1995.

[5] L. A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Transactions on Software Engineering*, SE-2(3), Sept. 1976.

[6] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. In *IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 522–525, 1992.

[7] L. Dillon and Y. Ramakrishna. Generating oracles from your favorite temporal logic specifications. In *ACM SIGSOFT Foundations of Software Engineering*. ACM Press, Oct. 1996.

[8] L. Dillon and Q. Yu. Oracles for checking temporal properties of concurrent systems. In *ACM SIGSOFT Foundations of Software Engineering*, pages 140–153. ACM Press, Dec. 1994.

[9] M. B. Dwyer and L. A. Clarke. Data flow analysis for verifying properties of concurrent programs. In *Proceedings of the 2nd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 62–75, Dec. 1994.

[10] N. Gupta, A. Mathur, and M. L. Soffa. Automated test data generation using an iterative relaxation method. In *Proceedings Foundations of Software Engineering*. ACM Press, Nov. 1998.

[11] G. Holzmann. Designing executable abstractions. In *Proceedings of the 2nd Workshop on Formal Methods in Software Practice*, pages 103–108, Mar. 1998.

[12] G. J. Holzmann. The model checking SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[13] P. Koppol and K.-C. Tai. An incremental approach to structural testing of concurrent systems. In *Proceedings of the 1996 International Symposium on Software Testing and Analysis*, pages 14–23. ACM Press, Jan. 1996.

[14] B. Korel. Automated software test generation. *IEEE Transactions on Software Engineering*, 16(8):870–879, Aug. 1990.

[15] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Boston, 1993.

[16] T. O'Malley, D. Richardson, and L. Dillon. Efficient specification-based test oracles. In *Second California Software Symposium*, Apr. 1996.

[17] D. J. Richardson. Testing with analysis and oracle support. In *Proceedings of the 1994 International Symposium on Software Testing and Analysis*. ACM Press, Aug. 1994.

[18] R. N. Taylor, D. L. Levine, and C. D. Kelly. Structural testing of concurrent programs. *IEEE Transactions on Software Engineering*, 18(3):206–215, Mar. 1992.

[19] S. N. Weiss. A formal framework for the study of concurrent program testing. In *Proceedings of the Second Workshop on Software Testing, Verification, and Analysis*, pages 106 – 113. IEEE Computer Society Press, July 1988.

# Testing 3

# Test Automation for Object-Oriented Frameworks

**Moritz Schnizler, Horst Lichter**
Department of Computer Science
Aachen Technical University
D-52056 Aachen
{moritz, lichter}@informatik.rwth-aachen.de

Key Words: Testing, Object-Orientation, Collaboration, Framework, Program Family

## 1 Introduction

Testing is one of the most important activities in the software development process. Only a thoroughly tested program will possibly fulfill the user's expectations. Even a systematic and careful development process can not prevent the need for final testing, see for example [Dyer 1992]. Consequently a product has to pass through an appropriate and carefully planned test, before it is released to the public.

Test Automation has the benefit that test cases once developed, can be reused in an eventual regression test. On one hand, this is essential during product maintenance, when corrections or changes have been made and developers have to verify that nothing else was broken. On the other hand test automation is useful for testing program families which are recently gaining importance in form of product lines [Weiss & Lai 1999].

## 2 Framework Test Bench

Program families, as defined by [Parnas 1976], are a set of programs, where it is worthwhile to first study their common properties, before determining the special properties of the individual family members, also called program variants. In other words, the members of a program family share the same implementation core, but actually represent program variants for e.g. different platforms, application areas and customers.

Object-oriented frameworks are an ideal means for developing program families. Actually an object-oriented framework represents an "abstract design" [Johnson & Foote 1988]. It comprises many design decisions and can be extended into a complete application. Today many projects use object-oriented framework technology for the development of program families, see for example [Bäumer et al. 1997].

While the use of framework technology increases productivity, testing the individual members of a program family remains laborious. So individual members of a program family are tested with limited or no reuse of test cases. Considering that all members of a program family have the same common core, this seems to be unnecessary. Test cases, which retest the common functionality of different family members, should be easily reusable form one member to the next.

To tackle this problem, we propose the concept of a test bench for program families which is adapted to a particular program family. Comparable to test benches from other engineering areas, e.g. for engines in automotive engineering, the test bench automates testing the common parts of a program family. Furthermore the test bench can be extended for the requirements of a particular program variant. This test bench itself is based on a test bench framework, see figure 1, containing the essential infrastructure for test automation. To adapt this test bench framework to the program family under test, test cases, which are specific for the program family, have to be implemented on top of this framework.

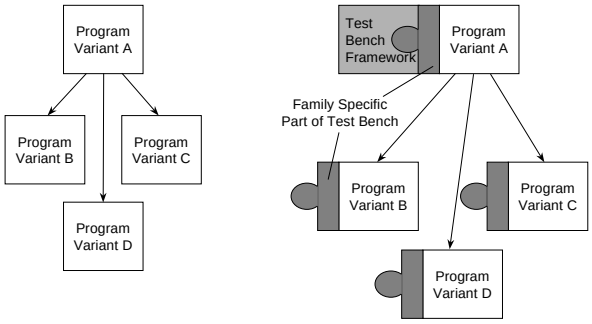



**Figure 1** Program Family without/with Test Bench

Because the common properties of a program variant are implemented using framework technology, technically the test bench is adapted to the domain-specific framework beneath the program family. In fact a framework specific test bench is realised. If a program family is based on more than one domain framework, their test benches can be combined, assuming they are based on the same test bench framework.

Because the test bench concept dramatically improves reuse of test cases, it allows thorough regression testing what is important for program families and frameworks. Actually a test bench serves two purposes: First of all, it is impossible to test generic elements in a framework having no concrete implementation. Secondly developers will introduce new errors when they adapt generic elements for their purposes. Both problems are alleviated, if it is possible to make a regression test of any concrete adaptation.

## 3 Test Cases for a Test Bench

A main issue with this approach is the question, what kinds of test cases are best suited for integration in the test bench? In this section we propose some properties, such test cases should have, and take a respective look at current testing techniques for object-oriented software.

### 3.1 Test Case Properties

We identified the following properties to be important for test cases, which can be integrated in an appropriate test bench:

- *Abstraction*: We can not test everything. Test cases should be focussed on the externally visible behaviour of the framework under test.
- *Relevance:* While test cases should abstract from details, they should still be relevant enough to adequately test the framework's functionality.
- *Stability:* Test cases should be robust not breaking from small changes in the implementation. Otherwise the test bench approach would be too costly.
- *Scalability:* Frameworks can consist of a few classes solving one problem or hundreds of classes addressing various tasks. We need test cases for any granularity and want to combine them, if it is necessary.
- *Universality:* If we want to test some functionality, it must be possible to develop appropriate test cases. It is not tolerable that we can not derive test cases in some situations.

### 3.2 Brief Look at Current Techniques

Because this approach concentrates on testing object-oriented frameworks, we want to keep those issues in mind and take a brief look at current techniques for testing object-oriented software, we found in literature. Most work about testing object-oriented software concentrates on testing individual classes. There have been successful attempts to test individual classes using techniques from procedural programming [Fiedler 1989], based on state machines [Turner & Robson 1993, Hoffman & Strooper 1995, Binder 1999] or the abstract data type nature of objects [Doong & Frankl 1994].

All those techniques have in common that they view a single class as the central entity for testing. Within the scope of classes they produce stable and abstract test cases based on a class interface. But all techniques do not scale up well for interacting clusters of classes, because the underlying models get too complex. Another drawback of these techniques is the fact that they are usually restricted to certain types of classes and are therefore not universal.

Something we felt to be missing, are techniques for object-oriented integration testing. We found only one approach [Jorgensen & Erickson 1994] to test a complete subsystem that is not limited to a black-box test of the GUI. This approach is based on so called atomic system functions (ASF) which can be roughly described as the path of method calls, caused by some event at the system border and terminated by some output of the system.

Starting from the system border, the developed test cases are more abstract and universal than those at class level. But the implementation of the system is still considered, making those test cases relevant for testing critical functionality. Subsystems do not need a GUI for testing, making this approach quite scalable. A drawback is the stability of test cases, because they are tied up between the borders of the system and the internal implementation.

## 4 Testing Collaborations

In this section we will first explain our motivation and basic ideas for developing test cases from object-oriented collaborations, and subsequently how role modelling supports this effort. Afterwards we will describe a testing process for our approach and discuss where tools can help in automation of the involved tasks.

### 4.1 Collaborations and Testing

When proposing their ASF technique [Jorgensen & Erickson 1994] argue that traditional software development by functional decomposition stresses structure over behaviour which is one of the central elements of the object-oriented paradigm. They identify this as source for many problems arising, when traditional testing techniques are adapted for object-oriented systems.

While we also believe that it makes usually no sense to use traditional testing techniques for object-oriented systems, we have identified a source of problems in the object-oriented paradigm itself. As [Booch 1994] illustrates, object-orientation stresses decomposition into objects over algorithmic decomposition. As a matter of fact, object-oriented design methods allow for detailed description of structural relationships, for example using class diagrams. On the other hand the behaviour of a single object is fully specified by its class. But the collective behaviour of a group of objects comes in second position, if it is explicitly considered at all. We call such collective behaviour of a group of objects collaborations following the UML terminology, [Booch 1994] calls them mechanisms.

The statement, that collaborations are often not adequately specified, is supported by observations, which have been made in the maintenance phase of object-oriented systems [Wilde et al. 1993]. They identified distribution of program function across several classes, what is natural for object-oriented software, without proper documentation as a difficult problem that makes programs hard to understand. Because a behavioural description is the foundation for any test, consequently it also makes programs hard to test.

Our approach is to base tests on those collaborations that implement the essential functionality of an object-oriented system. In the case of a framework, this means developing test cases for those collaborations that define the externally visible and usable functionality of the framework. In short words, the extension points of the framework that can be used or extended by a program implemented on top of the framework, see also [Riehle & Gross 1998].

Behavioural design patterns, like for example *Observer* or *Chain of Responsibility* [Gamma et al. 1995], describe collaborations which have appeared valuable in various contexts. Collaborations can be composed like design patterns to achieve even more comprehensive collaborations. This is also possible for the respective test cases which can be combined to test the newly composed collaboration.

We believe using collaborations as basis for test cases gives us enough flexibility to integrate them into a test bench. They fulfill the following properties:

- *Abstraction:* They are well suited for abstraction from details, since they can be based completely on interfaces without touching implementation details.
- *Relevance:* Because we concentrate on externally visible collaborations, they are by definition relevant to adequately test the framework's functionality.
- *Stability:* Depending on the level of abstraction used to describe collaborations, they are more robust to change than test cases for individual classes.
- *Scalability:* As mentioned above, collaborations and their test cases can be composed.
- *Universality:* Collaborations are the essence of object-oriented systems.

### 4.2 Separation of Concerns

As [VanHilst & Notkin 1996] state, appropriately chosen collaborations encapsulate fewer design decisions than classes and are therefore more stable with respect to evolution. But how do we find appropriate collaborations? Similar to [Riehle & Gross 1998] we believe that classes are not well suited to describe collaborations. A class implements the behaviour of a complete object that usually participates in more than one collaboration. For example in figure 2 object m acts as element in a list of data and as subject in an implementation of the observer pattern. It follows we need a higher level of abstraction than offered by classes to describe the participation of an object in different collaborations. We found role modelling, as described by [Reenskaug et al. 1996], is a good way to separate concerns - in this case collaborations – which are mangled in one class.

A role model describes a structure of collaborating objects with their static and dynamic properties. A role defines the position and responsibilities of an object that takes part in such a structure of collaborating objects. Role modelling is actually an abstraction process suppressing irrelevant objects and unnecessary details of objects. An object's role in context of a given collaboration, described by a role model, specifies only the necessary capabilities of the object in the given context.

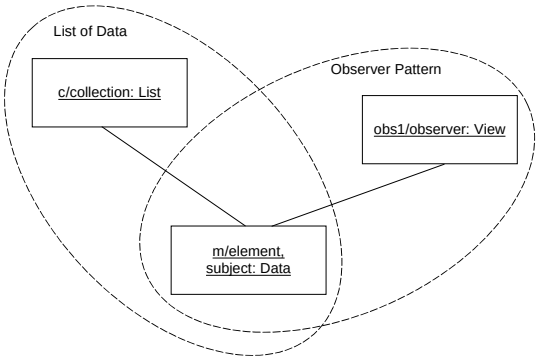



**Figure 2** Three objects in two collaborations

### 4.3 Example

Figure 3 shows the UML collaboration view of a role model describing the observer design pattern, as shown in figure 2. The role model abstracts from additional functionality of the object playing the subject role and possible other objects collaborating as observers for the same data. On the reverse side the collaboration view of the role model contains the information, necessary to describe the message sequence for updating all observing objects, in case the observed subject is changed.

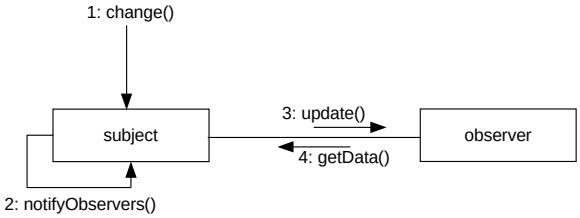



**Figure 3** Collaboration view of role model

Using the information from this diagram, test cases can be defined. As shown in figure 3, the trigger to start the update collaboration is the method change() that has to be implemented by the object playing the subject role. The test case is executed invoking this method for an object playing the subject role in a concrete instantiation of the role model, as shown in the UML object collaboration diagram in figure 4.

Because an abstract role model is not executable, we need to create instances of concrete objects for classes implementing the specified roles. For example in figure 4, the situation of one object of the class

Data playing the subject role and three prototypical objects of the class View playing the observer role is shown. Other test cases may require a different set up of object instances.

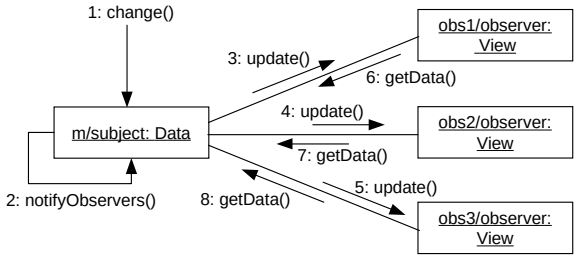



**Figure 4** Concrete instantiation of role model

To complete the test case, we need to determine an expected result to compare it with the actual result achieved by test execution. There are various possibilities to do so, depending on the goal of testing. An expected result can be defined by means of structural changes, for example the creation of a new observer object, changes in state of participating objects or parameter values for involved method calls. For the given example we could check, if all participating objects represent the same information after an update. Because sometimes we need to determine the state of an object, the code under test has to be extended by additional inspection methods.

However, as can be seen in figure 3, the collaboration view of the role model usually gives not enough information to specify expected results and therefore complete test cases. On one hand, we could use informal descriptions to substantiate the role model, but this would make tool support for test case generation difficult. Contracts [Helm et al. 1990] are a more formal alternative allowing the detailed specification of obligations between collaborating objects. Another possibility is the use of the UML object constraint language (OCL) [OMG 1999] to enrich the role diagrams with additional information.

### 4.4 Process and Tools

In this section we explain our process to develop test cases for collaborations and the possibilities for tool support of the involved tasks. As shown in figure 5, the source code of the program or framework under test is the starting point for developing collaboration based test cases. In the first step the developer adds information about the roles a class implements to the source code. Such a role description must indicate, what operations of the class belong to the role and what are the other roles, it collaborates with. For example [Riehle 2000] gives some pseudo-Java notation for documenting such role models that can be adapted for this purpose. This information is integrated using structured comments, leaving the Java code semantically unchanged. As discussed above, it is also necessary to improve the testability of the code by implementing additional inspection methods to ease the realisation of more comprehensive test cases.

In a second step the extractor tool uses the given description of a role models static structure to extract only those methods of a class which are relevant to its respective role. Additionally, it analyses the code of the implemented methods collecting information about its dynamic behaviour – its collaborations. Both types of information are combined into an internal representation that can be used by other tools. For example in a third step, a visual editor allows visualisation of the extracted role model using for example UML representing its static structure and its collaborations. Further it allows the definition of additional constraints for the represented collaborations using OCL or a similar enhancement to the UML easing the development of test cases.

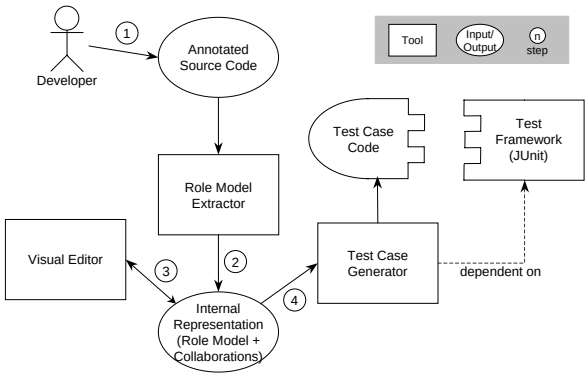



**Figure 5** Testing Process

Another tool that uses the internal representation of the role model is the test case generator. In a fourth step, it assists in the definition of test cases for the different collaborations of a given role model. For example, this tool suggests available collaborations, for which the tester can then create test cases by providing appropriate preconditions and expected results. Especially this tool assists in the set up of the necessary configuration of object instances for a specific test case. The test case generator is closely related to the test execution framework that finally executes the developed test cases. It generates Java code for the test cases and additional set up code according to the extension points of that framework.

One possibility for a test execution framework is the JUnit testing framework [Beck & Gamma 1998] that offers a simple, but flexible approach to implement and execute tests. In fact, in the end this framework is the test bench, mentioned above, while the test cases developed according to this process and which make finally part of the system under test as executable code are the program or framework specific part of the test bench.

## 5 Conclusions and Outlook

In the preceding sections, we showed that test automation makes sense for the development and maintenance of programs, and especially for program families. For this reason, we proposed our model of a

test bench for object-oriented frameworks, the basis of program families.

Following the need to realise test cases for a test bench, we examined the applicability of current techniques for testing object-oriented software. We showed that most of them depend too much on implementation details and scale up badly for clusters of collaborating classes. In contrast, we proposed the development of test cases focussing on object-oriented collaborations which can be specified using role modelling. We showed, how role modelling can be used to abstract from too many details and to separate concerns between different collaborations. While we showed that it is generally possible to develop test cases using role models of collaborations, it was also mentioned that, especially for automation of test case generation, more powerful means to specify obligations between roles have to be used. Currently we are evaluating different possibilities for introducing additional constraints into role models of collaborations, making them more suitable for test case generation.

Finally we described a process and associated tools for test case development and test automation according to our approach. After implementing some experimental versions of the test bench approach using the JUnit [Beck & Gamma 1998] testing framework as test execution framework, we are currently developing prototypes of the mentioned role model extractor and test case generator tools to gain more knowledge about the advantages and limitations of our approach.

## References


Bäumer, D., Gryczan, G., Knoll, R., Lilienthal, C., Riehle, D., Züllighoven, H. (1997): Framework Development for Large Systems, Communications of the ACM, vol. 40, no. 10, pp. 52 - 59, October, 1997.

Beck, K., Gamma, E. (1998): Test Infected: Programmers Love Writing Tests, Java Report, vol. 3, no. 7, pp. 40 - 50, July 1998.

Binder, R. (1999): Testing Object-Oriented Systems: Models, Patterns, and Tools, Addison-Wesley, 1999.

Booch, G. (1994): Object-Oriented Analysis and Design with Applications, 2 ed., Benjamin Cummings, 1994.

Doong, R.-K., Frankl, P. G. (1994): The ASTOOT Approach to Testing Object-Oriented Programs, ACM Trans. on Software Engineering and Methodology, vol. 3, no. 2, pp. 101 - 130, April 1994.

Dyer, M., The Cleanroom Approach to Quality Software Development , Wiley,1992.

Fiedler, S. P. (1989): Object-Oriented Unit Testing, HP Journal, vol. 40, no. 2, pp. 69 -74, April, 1989.

Gamma, E., Helm, R., Johnson, R., Vlissides, J. (1995): Design Patterns - Elements of Reusable Object-Oriented Software, Addison-Wesley, 1995.

Helm, R., Holland, I. M., Gangopadhyay, D. (1990): Contracts: Specifying Behavioral Compositions in Object-Oriented Systems, ACM SIGPLAN Notices, vol. 25, no. 10, pp. 169 - 180, October 1990.

Hoffman, D., Strooper, P. (1995): The Testgraph Methodology: Automated Testing of Collection Classes, JOOP, vol. 8, pp. 35 - 41, November/December 1995.

Johnson, R. E., Foote, B. (1988): Designing Reusable Classes, JOOP, vol. 1, no. 2, pp. 20 - 30; 35, June/July, 1988.

Jorgensen, P. C., Erickson, C. (1994): Object-Oriented Integration Testing, Communications of the ACM, vol. 37, no. 9, pp. 30 - 38, September 1994.

OMG (1999): Unified Modeling Language Specification, Version 1.3, June 1999.

Parnas, D. L. (1976): On the Design and Development of Program Families, IEEE Transactions on Software Engineering, vol. 2, no. 1, pp. 1 - 9, March 1976.

Reenskaug, T., Wold, P., Lehne, O. A. (1996): Working With Objects, Manning Publications Co., 1996.

Riehle, D., Gross, T. (1998): Role Model Based Framework Design and Integration, Proceedings OOPSLA '98, pp. 117 - 133, ACM Press, 1998.

Riehle, D. (2000): Framework Design - A Role Modeling Approach, PhD. Thesis, ETH Zürich, 2000.

Turner, C. D., Robson, D. J. (1993): The Testing of Object-Oriented Programs, University Durham, Technical Report TR-13/92, February 1993.

Weiss, D. M., Lai C. T. R. (1999): Software Product-Line Engineering - A Family-Based Software Development Process, Addison Wesley, 1999.

Wilde, N., Matthews, P., Huitt, R. (1993): Maintaining Object-Oriented Software, IEEE Software, vol. 10, no. 1, pp. 75 - 80, January 1993.

VanHilst, M., Notkin, D. (1996): Using Role Components to Implement Collaboration-Based Designs, Proceedings of OOPSLA '96, 1996.

# Object-Oriented Specification-Based Testing Using UML Statechart Diagrams

**Marlon E. Vieira** **Marcio S. Dias** **Debra J. Richardson**

*Information and Computer Science*
*University of California, Irvine*
*Irvine, CA 92697-3425 USA*
*{mvieira, mdias, djr}@ics.uci.edu*

## Abstract

This paper summarizes an approach to specification-based testing that employs UML statechart diagrams as the underlying specification, and automatically generates test drivers and a test script to execute the component under test. The test drivers cover the specification according to a selected test criterion and verify the behavior of the component under test. This approach has been implemented in a prototype called Das-Boot (Design And Specification-Based Object-Oriented Testing). The project has two short-term goals: 1) to define improved specification-based coverage criteria suitable for testing object-oriented software systems whose behavioral specification is modeled as a statechart; and 2) to develop techniques for generating test drivers, incorporating test cases and test oracles, with little interaction required by the human tester.

## Keywords

Specification-based testing, Test automation, Test oracle, Object-oriented, UML, Statecharts

## 1. Introduction

An increasing number of software systems have been developed using object-oriented technology. Object-oriented development emphasizes the creation of detailed models, which improve understanding of the software before it is actually built. These models also provide valuable information for the testing process by way of specification-based testing. Yet in spite of the high frequency of defects in both specifications and code and the costly consequences of these defects, there are few techniques for specification-based testing [1], especially ones that are applicable to commonly used means of specification.

The approach presented in this paper automates testing of Java classes based on their specifications represented as UML statechart diagrams. The Unified Modeling Language (UML) [2] is an evolution of previous object-oriented modeling languages and techniques and has been officially adopted as the OMG standard [3]. UML is a highly visual modeling language; in addition to words and text, it also consists (and in fact primarily consists) of graphs and diagrams. Additionally, UML is an inherently discrete language, meaning that it emphasizes discrete representations of dynamic behavior over continuous representations. The UML statechart diagram shows sequences of states that an object or an interaction may go through during its lifetime in response to received stimuli, together with responses and actions; therefore, it discretely represents potential dynamic behavior. The semantics and notation used in the UML's statechart diagrams are substantially those of David Harel's Statecharts [4] with modifications to make them object-oriented.

## 2. Background

Das-Boot employs the tenets of both specification-based testing, whereby information from the specification is used to guide the testing process, and state-based testing, which has traditionally focused on testing implementations whose structure reflects a finite state machine.

### 2.1. State-Based Testing

State-based testing has been considered by several researchers [5,6,7,8,9,10,11,12,13, 14,15]. The main purpose of these works has been to develop methods for generating a set of test cases, sometimes called a “test suite”, based upon a finite state machine (FSM) representation of the code, with the following objectives:

- Test suites should be relatively small – that is, there should be relatively few test cases;
- Each test case should be fast and easily executable in relation to the component under test;
- As many defects as possible in the [specification and/or] implementation are detected.

The FSM-based test generation methods differ in the compromises made between these conflicting objectives and the level of formalism upon which the method is based. Sidhu divides the methods for test generation based on FSMs broadly into two categories [16]: (1) those that rely on the use of characteristic sequences of the states in the FSM – examples in this category are the U-method, D-method, W-method and their variants – and (2) those that seek to construct test sequences for the actual behavior of an entity when it interacts with peer entities – these methods are based on state space exploration.

Object-oriented systems are well suited to use finite state machines to model their behavior [17] and consequently also befitting state-based testing. An object's state is defined as the combination of its attribute values. O-O state-based testing must, therefore, focus on an object's state-dependent behavior rather than the control and/or data structure.

Das-Boot generates a test suite for the implementation that traverses various sequences of states based upon the specification. Thus, Das-Boot's method falls into Sidhu's first category, because the test suite is generated based upon sequences of states for the objects. In addition, Das-Boot verifies whether the implementation's behavior is consistent with the statechart specification, detecting failures of the implementation. It does so by producing a test driver for each test case that executes the test and checks the object's attribute values as it transitions between states to determine that the correct transitions are taken and the correct state is reached, where correctness is determined by the statechart specification.

## 2.2. Specification-based Testing

Specification-based testing uses information derived from a specification to assist testing. When formal specification is used, it is possible to automate specification-based testing through the semantics of the specification formalism. Specification-based testing includes test case generation from specifications and/or test oracles derived from specifications whereby test results are compared with those specified. There are other uses of specifications in testing, including for instance evaluating testability based upon the specification. Das-Boot automates both aspects of specification-based testing by employing statechart specifications.

At least two artifacts are used for specification-based testing: (a) the specification model, which embodies the knowledge about the software requirements, and (b) the method used for test suite generation, including a coverage criterion describing how the specification will be covered, thereby guiding test execution.

Using specification-based testing to test the specification itself, is straight-forward, because the artifact upon which test cases are derived and/or coverage is measured is the same as the artifact being tested. Specification-based testing could be used, for instance, in driving simulations of a statechart specification, as is done in Argus-I [18].

On the other hand, using specification-based testing to test the implementation requires two further artifacts: (c) the implementation (component code) to be tested, and (d) a representation mapping between the specification and the implementation, which associates or maps the test requirements to the implemented component under test.

Object-oriented systems are well-suited to exploit specification-based testing because object-oriented development emphasizes the creation of detailed models prior to implementation, and some of these models have formal semantics that can be utilized to drive test generation.

## 3. DAS-BOOT's Approach

The testing process under Das-Boot's direction begins with the tester indicating the Java class to be tested and the statechart specification describing the desired class behavior. Based upon this information, the tester defines the representation mapping by associating the code to the specification. The tester then chooses a test coverage criterion. Based upon these selections, Das-Boot automatically produces test drivers (with embedded test oracles) to satisfy the criterion and compiles/executes the test drivers under a test script. At the end of this process, failures detected by the test oracles as discrepancies between behavior and the statechart specification are presented to the user. Figure 1 illustrates this process. In the following paragraphs, we further explain each step in this process, highlighting our research efforts.

### Statechart Specifications

Das-Boot loads XMI files for the statechart specifications. XMI (XML Metadata Interchange) is an OMG standard for the interchange of UML information between tools. XMI support gives Das-Boot the capability to read statechart diagrams from other UML modeling toolkits like Rational Rose [19] and Argo/UML [20]. We do not go into the detail of statecharts in this paper, but rather encourage the reader to look elsewhere.

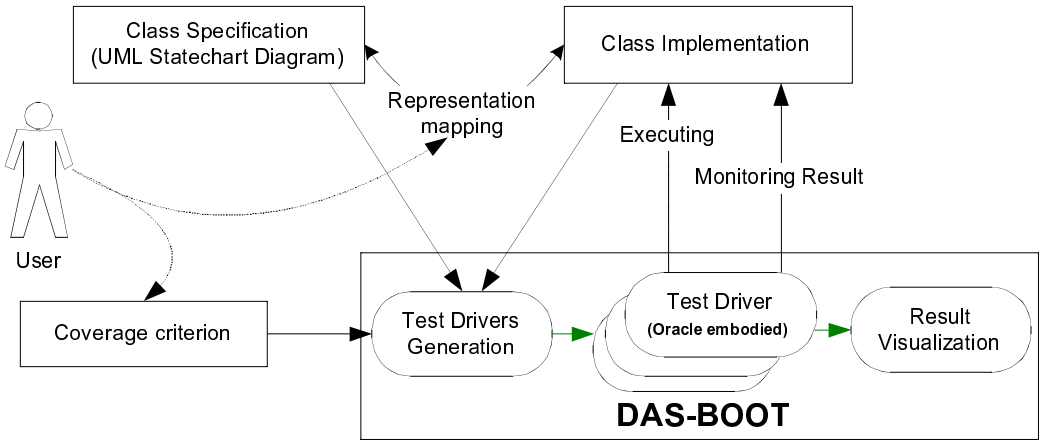



Figure 1. DAS-BOOT Process

## Representation Mapping

Das-Boot assists the tester in defining the representation mapping. The mapping is done by associating the following:

- statechart transitions –> Java class operation(s);
- statechart states –> Java class attributes(s);
- limits of a specific object state –>range of attribute values.

Adding this test information, which is sometimes called "making test-ready", requires additional work, but is by far easier than creating new specifications or models just for the purpose of testing. Although some might view this as a drawback of Das-Boot's approach, we view it as a major advantage, because augmenting with testing information enables specification-based testing to use existing models, potentially imported from another tool. This significantly reduces the overhead involved in using specification-based testing.

## Test Drivers

Das-Boot automates the actual testing process as much as possible by generating automated test drivers for each test case that not only force required coverage but also check to ensure that the implementation behaves according to the specification or model. Thus, for each transition (consisting of a source state, triggering event, action to be taken, and destination state) to be covered, the test driver developed puts the object into the source state, creates the circumstances leading to the triggering event, observes any actions taken and the destination state, and compares those actions and destination to those specified in the model. A point of our research is to make this generation more efficient, with little interaction required by the human tester. Currently, we are using templates to generate the test drivers and are evaluating this approach so as to improve them.

## Test Coverage Criterion

A test coverage criterion typically defines structures of the component under test that must be covered to satisfy the criterion; they are traditionally implementation-based and sometimes called structural criteria. For instance, branch coverage requires that every branch in the component under test be executed by at least one test case. The FSM-based criteria discussed above require that every state, transition, or traversal through the FSM model of the implementation is covered by execution of the test suite; as such they are implementation-based. Specification-based test coverage criteria define what to test based upon covering a behavioral specification of the component under test. Many specification-based criteria are analogous to implementation-based criteria extended by applying the core ideas to a specification rather than code. Das-Boot supports specification-based test coverage criteria that extend FSM-based test coverage criteria.

Das-Boot implements what we call the *StateVisited criterion*, which permits the tester to determine how many times a state is visited and when and how often a particular state is revisited; the tester typically makes this selection based on the granularity (range of values) associated with the state. The StateVisited criterion extends the W-method [21], which requires covering the tree of possible sequences of transitions from the initial state without revisiting states, the W-method is somewhat analogous to the implementation-based "basis path coverage" criterion. The StateVisited criterion allows revisiting a state after a transition sequence is exercised. We developed the StateVisited criterion because we believe that some failures could be revealed only when states are multiply visited. This criterion is useful first as a power-assist that enables the tester to increase the coverage performed by state-based testing. When

the tester decides to use the StateVisited criterion, however, the computational complexity is higher than the W-method. In spite of this, some points should be observed:

- Normally, the number of states modeled for a class is not too large, so that visiting states more than once imposes little increased complexity;
- The automation of the test tree construction and related test case generation and execution do not diminish the complexity problem, and may actually minimize it;
- Some classes in object-oriented software are more critical than others, thus requiring more extensive testing and greater coverage; this should be taken into account in applying the StateVisited criterion to each class.

## 4. Related Works

Recently, a number of techniques have been proposed for applying FSMs to object-oriented class testing. These state-based approaches to class testing concentrate on the interaction between the attributes and the member functions of classes. This work has shown that FSMs can be effectively used to test this interaction by representing the values of attributes as the states of FSMs and the member functions as the transitions of FSMs. Our work is related to approaches that apply UML statechart diagrams to class testing. Here we overview three approaches. Kim et al. [22] apply state diagrams to UML class testing: their approach flattens the hierarchical and concurrent structure of states and eliminates broadcast communications to obtain an extended finite state machine (EFSM); they also transform the resulting EFSM into a flowgraph from which control flow paths are generated and conventional data flow testing is applied. Kung et al.[23] describe a class testing technique using a variation of Statecharts, called object state diagram (OSD): their approach extracts and OSD directly from source code and generates test cases by constructing a spanning tree for the OSD. In contrast, we regard UML state diagrams as class specifications and propose a hierarchy of coverage criteria for UML state diagrams based on control and data flow information. Liuying and ZhiChang [24] present a method based on the Wp-method that automatically generates and selects test cases from UML: the Wp-method is an extension of the W-method, which uses partial characterization set instead of the entire characterization set, but guarantees complete fault coverage.

## 5. Conclusions and on-going work

We are developing an approach for testing the behavior of Java classes based on and against UML statechart diagrams; Das-Boot is a prototype tool that supports this approach. In conjunction with this, we are working on a comprehensive specification-based testing approach for object-oriented software systems that provides support from unit testing through integration and system testing by exploiting a variety of UML diagrams. Das-Boot serves as a testbed for experimenting with approaches to specification-based object-oriented testing.

A significant point in our research is to define better specification-based coverage criteria suitable for components whose behavioral specification is modeled as a statechart, and more broadly for systems whose requirements and design are specified using object-oriented notations. A widely used test coverage criterion for FSM-based testing is Chow's W method [21]. This method is not ideal to work with statecharts, however, due to the fact that statecharts extend traditional finite state machines with hierarchy and concurrence. The W-method is unable to cover the possibilities for parallel states. We are attempting to attack this problem in two directions. The first is by redefining our extension of the W-method to work on statecharts. Our starting point is to test every statechart transition through the combination of concurrent states – that is, to design test cases to set each possible current state, creating the circumstances which lead to an event, to observe the action taken, the transition made, and the new state(s). As there is a defined set of transitions in the state model, a coverage measure can be associated with the proportion of transitions exercised by a set of test cases. The second direction is to translate from statecharts to basic finite state machines, and then apply our extension of the W-Method. This direction is in line with our interest in applying model checkers, such as SPIN and SMV over statecharts, since existing model checkers work on traditional FSM representations.

It is important to point out the following efforts are yet to be considered, which may also be considered as future work: an empirical study to verify the effectiveness of the StateVisited test coverage criteria; and determination of the computational complexity related to the StateVisited criteria.

## References


[1] R.M. Poston, “Automating Specification-Based Software Testing”, IEEE Computer Society Press, ISBN 0-8186-7531-4, May 1996.

[2] Grady Booch, James Rumbaugh, Ivar Jacobson, “The Unified Modeling Language User Guide”, Addison-Wesley, 1999.

[3] OMG Unified Modeling Language Specification (draft), Version1.3 alphaR2, January 1999 (http://www.rational.com/uml/resources/documentation)

[4] D. Harel. Statecharts: A Visual Formalism for Complex Systems, “Science of Computer Programming, 8, 231-274 “-1987

[5] F. C. Hennie, “Finite-State Models for Logical Machines”, John Wiley & Sons, 1968.

[6] H.K.T. Ma, S. Devadas, A.R. Newton, and A.S- Vincentelli, “Test Generation for Sequential Circuits”, IEEE Trans. Computer-Aided Design, pp. 1,081-1,093, Oct. 1988.

[7] K.K. Sabnani and A.T. Dahbura, “A protocol testing procedure” Comput. Network and ISDN Syst., Vol 17, no4, pp285-297, 1988

[8] A. Ghosh, S. Devadas, and A.R. Newton, “Test Generation and Verification for Highly Sequential Circuits,” IEEE Trans. Computer-Aided Design, pp. 652-667, May 1991.

[9] D. Pomeranz and S.M. Reddy, “On Achieving Complete Fault Coverage for Sequential Machines,” IEEE Trans. Computer-Aided Design, pp. 378-386, Mar. 1994.

[10] F. C. Hennie, “Finite-State Models for Logical Machines”, John Wiley & Sons, 1968.

[11] C. D. Turner, and D. J. Robson, “State-Based Testing and Inheritance”, Technical Report TR1/93, University of Durham, England 1993

[12] D. Hoffman, P. Strooper. “The Testgraph methodology: Automated testing of collection classes”, JOOP Magazine , Nov-December 1995

[13] R. Binder, “The FREE approach for system Testing”, Object Magazine, 5(9):72-81, 1996

[14] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, “A Test Strategy for Object-Oriented Systems”, Proceedings, The Nineteen Annual International Computer Software and Applications Conference. August 1995, IEEE Computer Society Press, Los Alamitos, Calif. 239-244.

[15] B. Tsai, S. Stobart, N. Parrington, “A Method for Automatic Class Testing (MACT) Object-Oriented Programs Using A State-based Testing Method”, EuroSTAR '97, November 1997.

[16] D.P. Sidhu, A. Chung and T.P. Blumer “Experience with formal Methods in Protocol Development”, ACM SIGCOMM Computer Communication Review, no2, pp81-101, 1992

[17] R. Binder. Testing Object-Oriented Systems: Models, Patterns, and Tools, The Addison-Wesley Object Technology Series, 1999

[18] M.Dias, M.Vieira, D. Richardson, “Software Architecture Analysis based on Statechart Semantics”, Submitted to The 4th International Software Architecture Workshop (ISAW-4), ICSE 2000.

[19] Rational Rose, 2000 http://www.rational.com/products/rose/index

[20] Argo/UML: Cognitive Support for Object-Oriented Design, version 0.6.2, Jason Robbins, April 1999 (http://www.ics.uci.edu/pub/arch/uml/)

[21] T.S. Chow, “Testing Design Modeled by finite-state machines” IEEE Trans. Software Eng. Vol 4, pp 178-186, Mar. 1978

[22] Y.G Kim, H.S Hong, D.H. Bae, and S.D. Cha, “Test cases generation from UML state diagrams”, IEE Proceedings: Software, 146(4), 187-192, August 1999.

[23] D. Kung, N. Suchak, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen, ``On Object State Testing,'' in Proceedings of Computer Software and Applications Conference, pp. 222_227, 1994.

[24] L. Liuying and Q. ZhiChang “Test Selection from UML Statecharts”, Proceedings of the 31st International Conference on Technology of Object-Oriented Language and Systems, 1998

# A Framework for Practical, Automated Black-Box Testing of Component-Based Software

**Stephen Edwards**
Dept. of Computer Science
Virginia Tech
660 McBryde Hall
Blacksburg, VA 24061-0106 USA
+1 540 232 5723
edwards@cs.vt.edu

**ABSTRACT**
This paper briefly sketches a general strategy for automated black-box testing of software components that includes: automatic generation of component test drivers, automatic generation of black-box test data, and automatic or semi-automatic generation of component wrappers that serve as test oracles. This research in progress unifies several threads of testing research, and preliminary work indicates that practical levels of testing automation are possible.

**Keywords**
test drivers, test oracles, self-checking software, test adequacy, integration testing, modular construction, built-in test

## 1 INTRODUCTION

Modular software construction through the assembly of independently developed components is a popular approach in software engineering. At the same time, component-based approaches to software construction highlight the need for detecting failures that arise as a result of miscommunication among components. In component-based software, a component's interface (or specification) is separated from its implementation and is used as a contract between the clients and the implementer(s) of the component [9]. In practice, failures in component-based systems often arise because of semantic interface violations among components—where one party breaks the contract. These errors may not show up until system integration, when they are more expensive to identify and fix. Even worse, internal violations may not be discovered until after deployment. As a result, component-based development increases the need for more thorough testing and for automated techniques that support testing activities.

This paper outlines a general strategy for automated black-box testing of software components. The strategy is a three-pronged attack, covering automatic generation of component test drivers, automatic generation of test data, and automatic or semi-automatic generation of wrappers serving the role of test oracles. This work unifies several threads of testing research into a coherent whole. While many interesting and tough research questions remain open, preliminary results suggest practical levels of automation are achievable for components that include formal behavioral descriptions.

Section 2 describes the assumptions about components that are necessary for the approach to work, and presents an example component specification satisfying these assumptions. Section 3 discusses a critical piece of the strategy presented here: the use of pre- and postcondition checking wrappers around the component under test. Building on this foundation, Section 4 lays out the vision for an automated testing framework. Section 5 briefly discusses related work, followed by open research issues and future directions in Section 6.

## 2 AN EXAMPLE COMPONENT: ONE-WAY LIST

For the proposed strategy to work, what assumptions are made about components? First, a component must have a well-defined interface that is clearly distinguishable from its implementation. Second, in order to automate the process of generating test data or checking test results, one must have some description of the intended behavior of the component under test. The initial requirement for the research described here is that a component must have a formally specified interface described in a model-based specification language. RESOLVE [16] has been selected as the specification language for this research, although other model-based specification languages [17] are also applicable. The choice of specification language was made for two pragmatic reasons: the researchers involved were familiar with the language, and using it provides a natural collaboration path for fielding tools. Researchers at The Ohio State University and at West Virginia University are collaborating on a Software Composition Workbench based on RESOLVE technology that is an ideal environment in which to evaluate and apply the testing tools described in this paper.

```
concept One_Way_List
    context
        global context
            facility Standard_Boolean_Facility
        parametric context
            type Item
    interface

        type List is modeled by (
             left  : string of math[Item],
             right : string of math[Item]
           )
            exemplar s
            initialization
                ensures     s = (empty_string, empty_string)

        operation Move_To_Start (alters s : List)
            ensures     s = (empty_string, #s.left * #s.right)

        operation Move_To_Finish (alters s : List)
            ensures     s = (#s.left * #s.right, empty_string)

        operation Advance (alters s : List)
            requires    s.right /= empty_string
            ensures     there exists x : Item
                            (s.left = #s.left * < x >  and  #s.right = < x > * s.right)

        operation Add_Right (alters s : List,  consumes  x : Item)
            ensures     s = (#s.left, < #x > * #s.right)

        operation Remove_Right (alters s : List, produces x : Item)
            requires    s.right /= empty_string
            ensures     s.left = #s.left  and  #s.right = < x > * s.right

        operation Swap_Rights (alters s1 : List, alters s2 : List)
            ensures     s1.left = #s1.left  and  s1.right = #s2.right  and
                        s2.left = #s2.left  and  s2.right = #s1.right

        operation At_Start (preserves s : List) : Boolean
            ensures     At_Start  iff  s.left = empty_string

        operation At_Finish (preserves s : List) : Boolean
            ensures     At_Finish  iff  s.right = empty_string

end One_Way_List
```

**Figure 1—A RESOLVE Specification for One-Way List**

Although the initial research requirement is that all components have formally specified interfaces, the tools making up the approach do provide graceful fallback positions if only semi-formal or informal component behavioral descriptions are available. Informal descriptions require more human intervention in the process, however, since there is no easy way to automatically extract behavioral requirements. The end result is a strategy that can still be applied, even without any formal behavioral descriptions, but at the cost of reduced automation and greater programmer intervention.

To ground the discussion of formally specified components in this paper, Figure 1 presents the RESOLVE specification of a one-way list component that was originally described by Sitaraman *et al.* [15]. This generic component is parameterized by the type of `item` it will contain. A one-way list is an ordered sequence of items, all of the same type. One may move forward in the sequence, accessing individual elements in turn, or jump to either end of the sequence. A similar component that supports bi-directional movement is presented by Zweben [19]. A one-way list may be implemented as a singly-linked chain of

dynamically allocated nodes, as a dynamically allocated array, or by building on other components like a stack, a queue, or a vector.

The mathematical model of the one-way list shown in Figure 1 is a pair of mathematical strings (finite sequences). There is no explicit notion of a "current position" or "cursor." Instead, the current location is implicit in the fact that the string is partitioned into left and right segments. Intuitively, items to the "left" are those that are "behind" the current location (closer to the front of the sequence), while those to the "right" are in front (toward the rear). The preconditions (`requires` clauses) and postconditions (`ensures` clauses) of each operation supported by the component are described in terms of this mathematical model. In postconditions, the pound sign (#) is used to refer to the incoming value of a parameter, rather than its outgoing value.

## 3 THE CENTRAL FOCUS: BUILT-IN TEST CAPABILITIES

The cornerstone of the automated testing framework is a micro-architecture for providing built-in test (BIT) support in software components. This architecture builds on current research in systematically detecting interface violations in component-based software [4]. In essence, each software component provides a simple "hook" interface (with no run-time overhead) that can be used in adorning the component with sophisticated BIT capabilities. Figure 2 illustrates this idea. Sophisticated "decorator" components (wrappers) that provide a number of self-checking and self-testing features can then be used to encase the underlying component.

The innovative properties of this strategy are:

- BIT wrappers are completely transparent to client and component code.
- BIT wrappers can be inserted or removed without changing client code (only a declaration need be modified). This capability does not require a preprocessor, and can be used in most current languages.
- When BIT support is removed, there is no run-time cost to the underlying component.
- Both internal and external assertions about a component's behavior can be checked.
- Precondition, postcondition, and abstract invariant checks can be written in terms of the component's abstract mathematical model [4], rather than directly in terms of the component's internal representation structure.
- Checking code is completely separated from the underlying component.

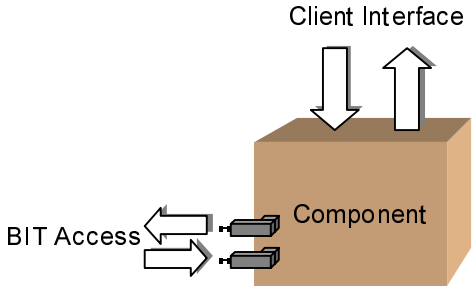



**Figure 2–Component Provides "Hooks" for BIT Infrastructure**

- Violations are detected when they occur and before they can propagate to other components; the source of the violation can be reported down to the specific method/operation responsible.
- Routine aspects of the BIT wrappers can be automatically generated.
- The approach works well with formally specified components, but does not require formal specification.
- The approach provides full observability of a component's internal state without breaking encapsulation for clients.
- Actions taken in response to detected violations are separated from the BIT wrapper code.

Figure 3 illustrates a component encased in a two-ply BIT wrapper. The inner layer of the wrapper is responsible for directly and safely accessing the component's internals, performing internal consistency checks, and then converting the internal state information into a program-manipulable model of the component's abstract state [4]. The outer layer is responsible for using this model to check that clients uphold their obligations in using the underlying component, to check that the component maintains any invariant properties it advertises, and to double-check the results of each operation to the extent desired for self-testing purposes. Client code accesses the component just as if it were unadorned.

The BIT strategy is designed to provide maximal support during unit testing, debugging, and integration testing. By outfitting a component with a BIT wrapper during unit testing, much more thorough testing can be achieved with the ad hoc strategies most developers employ. For every test case executed, a large number of internal consistency checks are performed, any one of which has the potential of revealing errors. Since these checks are automatically performed for any and all operations executed by the component in each test case, they have the effect of multiplying the tester's ability to detect errors. When errors are found, the full visibility of internal state provided by the BIT strategy is helpful during debugging. In particular, the

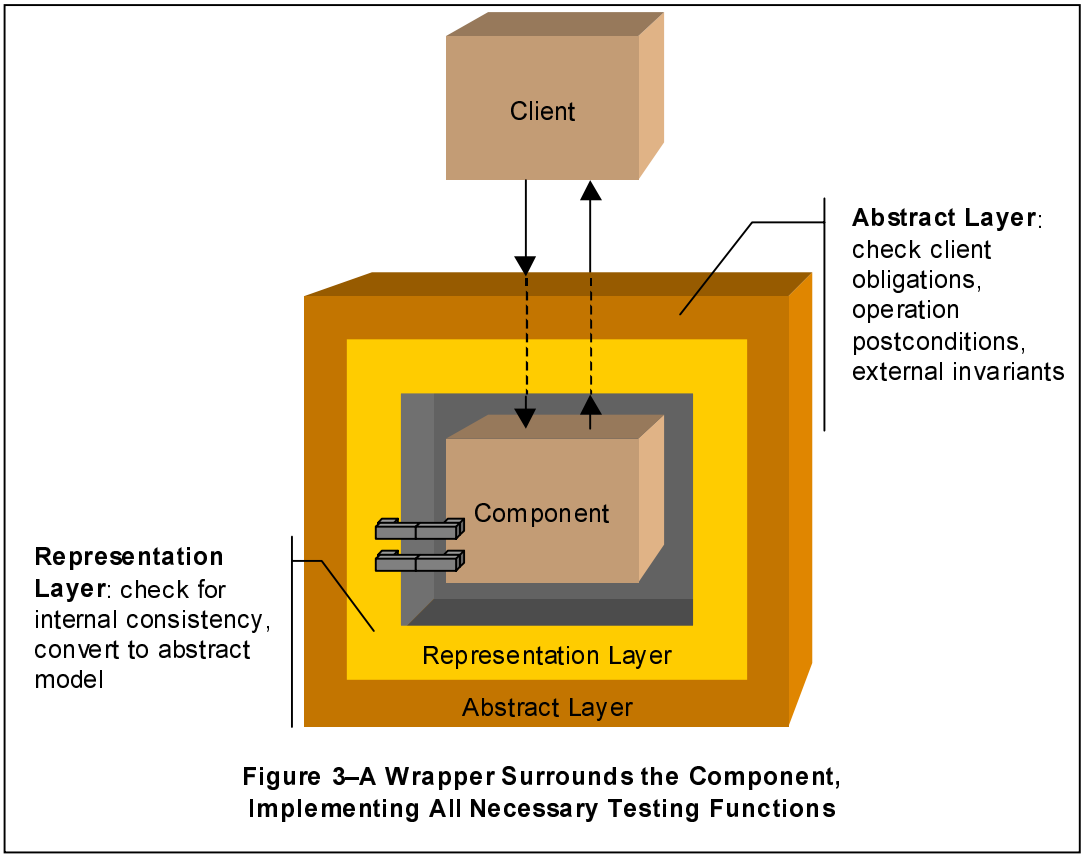



**Figure 3–A Wrapper Surrounds the Component, Implementing All Necessary Testing Functions**

strategy provides the programmer with additional capabilities for both input and output of internal state information, as well as the ability to modify internal state information for debugging purposes. None of these capabilities require any additional design or coding time from the developer, beyond the inclusion of the original BIT hooks in the underlying component. Finally, during integration testing, BIT wrappers can provide firewalls between components for incremental integration. As new units are added to the system, the wrappers will detect any unforeseen interactions. This strategy supports bottom-up, top-down, and hybrid incremental integration strategies.

## 4 THE VISION: AN AUTOMATED TESTING FRAMEWORK

The BIT infrastructure provides a natural mechanism for supporting semi- or fully automatic testing. Simply put, the framework for automated testing described here rests on three legs:

- Automatic (or semi-automatic) generation of a component's BIT wrapper.
- Automatic generation of a component's test driver.
- Automatic (or semi-automatic) generation of test cases for the component.

All three generation strategies rely on the same information: a complete behavioral description of the component's interface contract. By combining these generation strategies, it is possible to create a test driver, a test suite, and a BIT wrapper directly from a component's specification. If the BIT wrapper also provides comprehensive checks on the postconditions of all exported operations—in effect, acting as a test oracle—then the combination will produce a highly automated testing and debugging capability, as outlined in Figure 4.

### Generating BIT Wrappers

We have designed and implemented a generator that uses RESOLVE-style component specifications and C++ template interfaces to generate BIT wrappers [14]. The underlying principles for creating such wrappers are independent of any particular specification technique or implementation language, and they can be readily extended to other languages [4].

The external interface of a BIT wrapper is identical to that of the corresponding base component. Semantically, they differ in how they behave when either the pre- or postcondition of some operation is violated. In particular, where a regular component guarantees nothing if an operation is invoked under conditions violating its precondition, a BIT wrapper instead guarantees it will perform a specific notification action. We call a BIT wrapper that only checks for precondition violations a one-way checking wrapper.

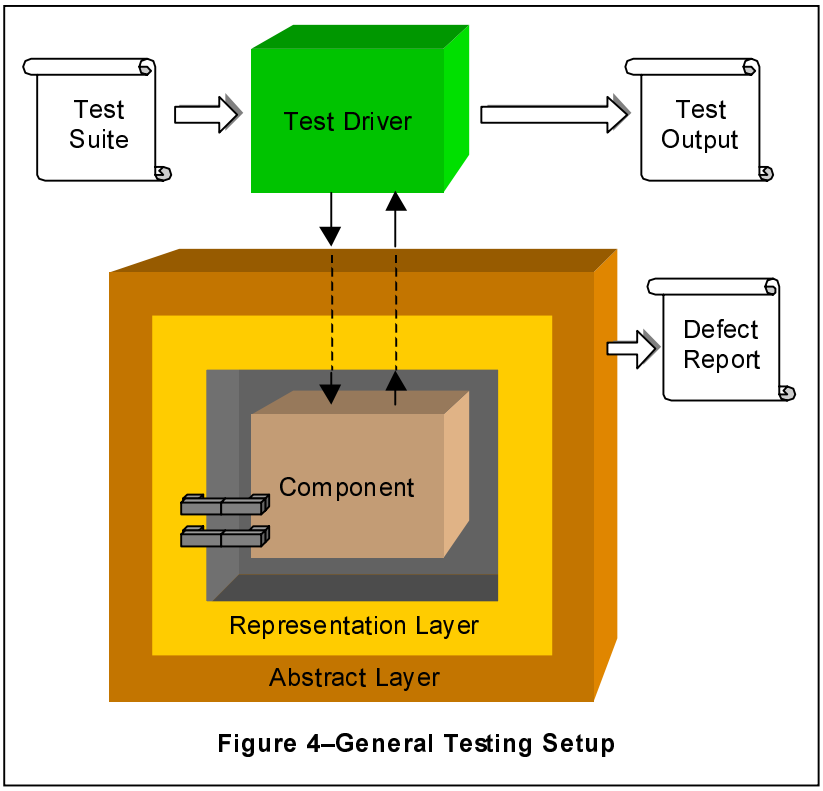



**Figure 4–General Testing Setup**

Similarly, a two-way checking wrapper guarantees to:

1. Carry out its precondition notification action if the precondition does not hold, or
2. Establish that the postcondition is true upon operation completion, or
3. Carry out its postcondition notification action if the postcondition does not hold.

Both one-way and two-way checking wrappers are extremely useful. One-way wrappers correspond with the traditional notion of a "defensive shell" that protects a component from errant clients. Two-way wrappers, on the other hand, are more akin to "self-checking" or "self-verifying" components that confirm their own work as well as spotting erroneous client behavior.

While constructing BIT wrappers is a straightforward process, it raises the question of how one can automatically generate pre- and postcondition checks. For most components, checking each precondition is straightforward and can thus be automated. By using the model conversion approach described in [4], many precondition and postcondition assertions can be converted to code by a simple transliteration process. For example, complete pre- and postcondition checks can be automatically generated for the one-way list specification in Figure 1.

However, some assertions are non-trivial. For example, code for checking assertions containing quantifiers cannot be generated mechanically [2]. As a result, we are exploring the following possibilities for providing greater support for automated BIT wrapper construction:

- **Semi-Automatic Generation**: It is possible to automatically generate checking code for many preconditions as well as for many clauses in postconditions. One possible approach to solving this problem is to automatically generate everything that is appropriate, and allow a human to provide the code for those checks that cannot be automated. Our experience with the prototype wrapper generator indicates it is a simple process to separate the human-contributed checks from all of the other infrastructural code necessary to support a BIT wrapper. Further, the person creating the checks will write them in abstract client-level terms—i.e., the mathematical model of the component's state—instead of in terms of the implementation of the component under test [4].
- **Dynamic Assertion Verification**: An alternative that we are actively exploring uses current generation verification tools. While current verification tools often have trouble with complex quantified assertions that arise during static formal verification, the simpler assertions that arise at run-time in a BIT wrapper, where all variables have specific values, are more amenable to existing proof tools. It is possible to automatically generate a complete BIT wrapper that relies on a verification/proof engine for assertion checking with specific parameter values at run-time.
- **"Armored" Components Using Reference Implementations**: If a reference implementation for a component (even an inefficient one) exists, it is possible to automatically generate a BIT wrapper that performs back-to-back testing against the component under test. This opens up intriguing possibilities, since it is possible to recover from internal errors; the wrapper, which stands between the client and the two implementations, can selectively pass on the reference implementation results when the unit under test fails. Further, the BIT infrastructure even allows the "good" data produced by the reference implementation to be used to force recovery on the unit under test [4]. This leads to a defensive wrapper that is close to bulletproof.

**Generating Test Drivers**

Compared to the difficulties involved in generating BIT wrappers, generating test drivers is a simpler problem. The research described here is based on an interpreter model for test drivers: a test driver can be viewed as a command

interpreter that reads in test cases and translates them into actions on the component under test. From this point of view, it is straightforward to parse a component's interface definition, identify its operations, and construct an interpreter. All filtering of invalid operation requests is handled by the BIT wrapper encasing the component under test, as is run-time checking of produced output. The major weaknesses of this approach are in effectively handling of components that rely on inversion of control or that have a substantial human interaction component.

We have designed and are currently implementing a test driver generator based on this strategy. We are currently using RESOLVE/C++ as the underlying implementation language for our components [16], and so have adopted a subset of C++ as a test case definition language. Figure 5 shows a sample test case one might use for the one-way list component.

The architecture for the interpreter/test driver uses the envelope and letter paradigm for handling internal values, and uses an exemplar-based dispatching strategy for handling operations on user-defined objects [3]. As a result, the core interpreter engine does not directly refer to the component under test or any of its methods. This means that support for any unit under test can be added without requiring any changes to or recompilation of the interpreter engine itself. Instead, our driver generator creates a "glue" source file that, when compiled and then linked with the existing interpreter object files, produces a custom driver for the component under test. Our experience has been that an interpreter provides a significant time savings over direct compilation of test cases when large test sets are used.

**Generating Test Data**

There are a number of strategies for generating black-box test data from a component's behavioral description [1]. The generation approach we have taken is adapted from black-box test adequacy criteria described by Zweben et al. [19]. This black box test adequacy work describes how one can construct a flow graph from a behavioral specification. This directed graph has a single entry, representing object creation, and a single exit, representing object destruction. Every "object lifetime"—composed of some legal sequence of operations applied to a given object—is represented as some (possibly cyclic) path through the graph.

Given such a flow graph, possible testing strategies become evident [1]. Zweben et al. describe natural analogues of white-box control- and data-flow testing strategies adapted to black-box flow graphs, including node coverage, branch coverage, all definition coverage, all use coverage, all DU-path coverage, and all k-length path coverage. Further, because branches in the graph represent different choices for method calls in a sequence, instead of logical control-flow decisions, it is easier to generate test cases that exercise all branches.

```
{
   List l;
   Integer x;
   x = 43751;
   l.Add_Right (x);
   l.Remove_Right (x);

   cout << "output => " << l
        << ' ' << x << endl;
}
```

**Figure 5—A One-Way List Test Case**

As with other black-box test generation strategies, this approach faces two open issues: how to correctly and efficiently decide which edges should be included in a graph, and how to address the problem of satisfiability in choosing test data values to be used in individual test cases. While perfect solutions to these problems are not computable, practical heuristics that provide approximate solutions are available. When combined with a BIT wrapper surrounding the component under test, invalid test cases can be automatically screened and removed, allowing overly optimistic heuristics to be used in practice. Further, the internal checks performed by BIT wrappers have the possibility of revealing defects that are not directly observable from the output produced by operations. This property can lead to an automated testing approach that has a greater defect revealing capability than traditional black-box strategies.

We have experimented with automatically generating test data by constructing flow graphs directly from RESOLVE specifications. We have implemented a prototype tool for this purpose, and are currently in the process of evaluating the effectiveness of the corresponding test data using fault injection techniques [18]. Specifically, we have taken four RESOLVE-specified components (a stack, queue, one-way list, and partial map), and applied an expression-selective mutation testing strategy [10] to seed known defects in them. Test sets for each component were then generated using the approach outlined here for three coverage criteria: all nodes, all definitions, and all uses. Each mutant was run on the corresponding test sets, and data were collected both with and without BIT wrappers.

The preliminary results of this experiment are summarized in Table 1. "# Test Cases" indicates the number of distinct test cases (such as the example in Figure 5) in each test set. "Output Failures" indicates the number of mutants killed by the corresponding test set based solely on observable output (without considering violation detection wrapper checks). "BIT Failures" indicates the number of mutants killed solely by using the invariant and postcondition checking provided by the subject's BIT wrapper. Figure 6 provides a graphical summary of the percentage of mutants killed

| Adequacy Criterion | Subject | # Test Cases | Output Failures | % | BIT Failures | % |
|---|---|---|---|---|---|---|
| All Definitions | Stack | 6 | 6 | 21.4% | 16 | 57.1% |
| | Queue | 6 | 5 | 18.5% | 15 | 55.6% |
| | One-Way List | 11 | 47 | 39.2% | 83 | 69.2% |
| | Partial Map | 6 | 92 | 37.6% | 132 | 53.9% |
| | Total | 29 | 150 | 35.7% | 246 | 58.6% |
| All Nodes | Stack | 5 | 16 | 57.1% | 22 | 78.6% |
| | Queue | 5 | 15 | 55.6% | 21 | 77.8% |
| | One-Way List | 10 | 37 | 30.8% | 89 | 74.2% |
| | Partial Map | 7 | 118 | 48.2% | 153 | 62.4% |
| | Total | 27 | 186 | 44.3% | 285 | 67.9% |
| All Uses | Stack | 32 | 26 | 92.9% | 28 | 100.0% |
| | Queue | 32 | 25 | 92.6% | 27 | 100.0% |
| | One-Way List | 126 | 106 | 88.3% | 120 | 100.0% |
| | Partial Map | 61 | 185 | 75.5% | 214 | 87.3% |
| | Total | 251 | 342 | 81.4% | 389 | 92.6% |

**Table 1—Expression-Selective Mutation Scores of Test Sets**

under each condition, together with the increased detection rate provided by BIT wrappers.

As expected, all uses coverage provided a higher defect-revealing capability than the other criteria. The use of two-way checking BIT wrappers provided an improvement in defect revealing capability in every case where they were used, ranging from 8%–200% more mutants killed. The greatest improvement was seen in the weakest test sets; for example, the all definitions test set for the queue component only revealed 18.5% of defects by examining test output alone, but this rate increased to 55.6% with a BIT wrapper.

Most surprising of all, for three of the four components, 100% of seeded defects were revealed by using BIT wrappers together with all uses coverage criteria. The fourth and most sophisticated component, the partial map, showed an 87.3% defect detection rate under the same circumstances. This result appears to be caused by two separate factors. First, the heavy degree of automated checking provided by the BIT wrappers significantly boosted the defect detect rate of the test sets. Second, the components that saw 100% defect detection rates had significantly fewer complex logic conditions and nested control constructs in their methods, and code instrumentation analysis revealed that full white-box statement-level coverage was being achieved by the test sets. This may be atypical of most components, so the 100% fault detection results for "all uses" should not be unduly generalized. On the other hand, design guidelines for object-oriented software argue for small, simple methods, even in complicated objects, so it is conceivable that the components used in this study are more representative of current coding practices than the deeply nested logic of procedural-style programming.

The preliminary results provided by this experiment, together with our experiences with the generator, indicate that there is the potential for practical automation of this testing strategy.

## 5 RELATED WORK

The BIT wrappers here are built on a philosophy perhaps best phrased by Bertrand Meyer as design-by-contract [9]: preconditions of operations are the responsibility of callers while postconditions are the obligations of implementers, and implementers may thus assume that the preconditions hold at the time of invocation. Others have proposed different allocations of responsibilities [8, 12]. One key difference in the approach advocated here is that responsibility for checking whether or not obligations are met should be separated from both client and implementer.

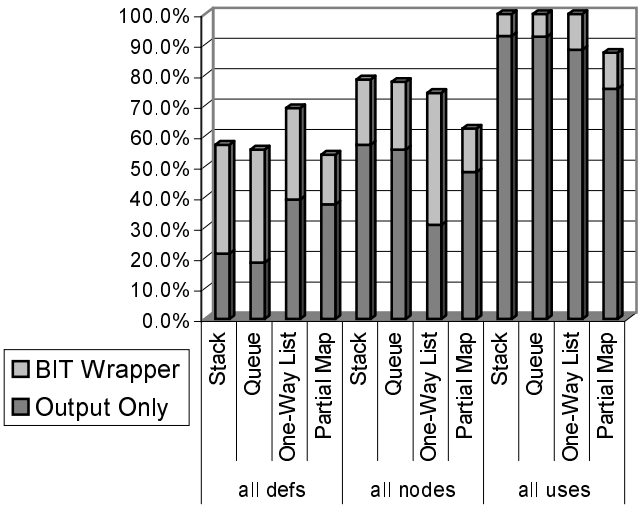



**Figure 6—Defect Detection Rates**

In addition to decoupling checking code from both the client and the component, this also opens up the opportunity of performing checks in client-level, abstract terms instead of in component-level implementation details. This results in highly reusable wrappers that easily can be added to or removed from a system.

Many others have also discussed the idea of run-time assertion checking. The Annotation Pre-Processor described by Rosenblum [13] is a good example. However, such approaches typically do not distinguish between the abstract view of component state perceived by clients and the concrete, implementation details seen by implementers. In addition, such approaches are rarely integrated into an overall strategy for automated testing. Eiffel provides another well-known approach for pre- and postcondition checking at runtime [9]. A more complete discussion of differences between BIT wrappers and Eiffel assertion checking is provided in [4], but the Eiffel approach is not combined with a systematic approach to producing test drivers or test data.

Other published approaches to specification-based testing of object-based and procedural software components [2, 5, 6, 7, 11] have influenced this work. The research described here differs, however, in the way it incorporates run-time interface violation checking, a strategy for generating test data, a design for unit and integration test drivers, and the way it separates testing infrastructure code completely from all units under test in a system.

## 6 CONCLUSIONS AND FUTURE WORK

This paper briefly sketches a general strategy for automated black-box testing of software components. The strategy is based on combining three techniques: automatic generation of component test drivers, automatic generation of test data, and automatic or semi-automatic generation of wrappers serving the role of test oracles. This research in progress unifies several threads of testing research into a coherent whole. Several difficult research questions remain open, but work to date indicates that practical levels of testing automation are possible.

The primary open research issues for future work include:

- Evaluating the effectiveness of test data produced using our current prototype tool.
- Exploring the limits of semi-automatic generation of postcondition checking code in BIT wrappers.
- Assessing the feasibility of dynamic verification of postconditions as an alternative implementation strategy for BIT wrappers.
- Exploring alternative heuristics for generating flow graphs from specifications.
- Exploring alternative heuristics for selecting specific data values to be used in generated test cases.
- Completing and evaluating the test driver generator.

## ACKNOWLEDGEMENTS

The ideas and feedback provided by members of the Reusable Software Research Group at The Ohio State University and at West Virginia University have helped shape and direct this research. In addition, several graduates students have contributed to the exploratory work described here: Vinay Annojjula, Sharmin Banu, Nikhil Bobde, Duxing Cai, Didem Durmaz, Rajat Gupta, Bob Hall, Mandar Joshi, Hunter Nuttal, Kent Swartz, Manoj Thopcherneni, and Wei Wang. Their contribution is gratefully acknowledged.

## REFERENCES

1. Beizer, B. *Black-Box Testing: Techniques for Functional Testing of Software and Systems.* Wiley, New York, 1995.
2. Bennett, B., and Sitaraman, M. Validation of results in testing abstract data types: A method for automation. In *Proc. 1st Int'l Conf. Software Quality* (Dayton, Ohio, Oct. 1991).
3. Coplien, J.O. *Advanced C++ Programming Styles and Idioms*. Addison-Wesley, 1992.
4. Edwards, S., Shakir, G., Sitaraman, M., Weide, B.W., and Hollingsworth, J. A framework for detecting interface violations in component-based software. In *Proc. 5th Int'l Conf. Software Reuse* (Victoria, Canada, June 1998) IEEE CS Press, 46-55.
5. Frankl, P., and Doong, R. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Software Eng. Methodology 3*, 2 (1994), 101-130.
6. Gannon, J.D., McMullin, P.R., and Hamlet, R. Data-abstraction implementation, specification, and testing. *ACM Trans. Programming Languages and Systems 3*, 3 (July 1981), 211-223.
7. Hoffman, D., and Strooper, P. The test-graphs metholodogy—Automated testing of classes. *J. Object-Oriented Programming* (Nov./Dec. 1995), 35-41.
8. Liskov, B., and Guttag, J. *Abstraction and Specification in Program Development*, McGraw-Hill, New York, 1986.
9. Meyer, B. *Object-Oriented Software Construction, 2nd Ed.*, Prentice Hall PTR, Upper Saddle River, New Jersey, 1997.
10. Offutt, J., Lee, A., Rothermel, G., Untch, R.H., and Zapf, C. An experimental determination of sufficient mutant operators. *ACM Trans. Software Eng. Methodology 5*, 2 (Apr. 1996), 99-118.
11. Parrish, A., and Cordes, D. Applying conventional unit testing techniques to abstract data type operations.

*Int'l J. Software Eng. and Knowledge Eng. 4*, 1 (Mar. 1994), 103-122.

12. Perry, D.E. The inscape environment. In *Proc. 11th Intl. Conf. On Software Eng.* (May 1989), IEEE CS Press, pp. 2-12.

13. Rosenblum, D.S. A practical approach to programming with assertions. *IEEE Trans. Software Eng. 21*, 1 (Jan. 1995), 19-31.

14. Shakir, G. *A Systematic Generator for Detecting Interface Violations in Component-Based Software.* M.S. Report, Dept. of Computer Science and Elec. Engineering, West Virginia Univ., Morgantown, WV, 1999.

15. Sitaraman, M., Welch, L.R., and Harms, D.E. On specification of reusable software components. *Int'l J. Software Eng. and Knowledge Eng. 3*, No. 2 (1993), 207-229.

16. Sitaraman, M., and Weide, B.W., eds. Component-based software engineering using RESOLVE. *ACM SIGSOFT Software Eng. Notes 19*, 4 (1994), 21-67.

17. Wing, J. M. A specifier's introduction to formal methods. *IEEE Computer 29*, 9 (Sept. 1990), 8-24.

18. Zhu, H., Hall, P.A.V., and May, J.H.R. Software unit test coverage and adequacy. *ACM Computing Surveys 29*, 4 (Dec. 1997), 366-427.

19. Zweben, S., Heym, W., and Kimmich, J. Systematic testing of data abstractions based on software specifications. *J. Software Testing, Verification and Reliability 1*, 4 (1992), 39-55.

# Toward the Determination of Sufficient Mutant Operators for C

**Ellen Francine Barbosa**
**José Carlos Maldonado**
**Auri Marcelo Rizzo Vincenzi**

Depto. de Ciências de Computação e Estatística
ICMC/USP – Universidade de São Paulo
Av. Dr Carlos Botelho, 1465
Cx. Postal 668,
13560-970 São Carlos, SP, Brazil
{francine, jcmaldon, auri}@icmc.sc.usp.br

**ABSTRACT**

Mutation Testing – one of the fault-based criteria – has been found to be effective on revealing faults. However, its high cost of application, due to the high number of mutants created, has motivated the proposition of alternative approaches for its application. One of them, named Selective Mutation, aims to reduce the number of generated mutants through a reduction on the number of mutant operators. A previous relevant study resulted on the proposition of a sufficient mutant operators set for FORTRAN, indicating that it is possible to have a large cost reduction of Mutation Testing, preserving a high Mutation Testing score. In the same research line, this work investigates procedures for the determination of a sufficient mutant operators set for C programs in the perspective of contributing to the establishment of low-cost, effective mutation based testing strategies.

**Keywords**

Software Testing, Mutation Testing, Sufficient Mutant Operators

## 1 INTRODUCTION

Software testing, which has as objective the identification of not-yet-discovered errors, is one of the most important activities to guarantee the quality and the reliability of the software under development. The success of the testing and validation activities depends on the quality of a test set.

Since the exhaustive test is, in general, impracticable, criteria that allow selecting a subset of the input domain preserving the probability of revealing the existent errors in the program are necessary. These criteria systematize the testing activity and may also constitute a coverage measure[9].

There are a large number of criteria available to evaluate a test set for a given program against a given specification. A tester may use one or more of these criteria to assess the adequacy of a test set for a program and, if it is the case, enhance the test set by constructing additional test cases needed to satisfy the selected criteria.

Considering the diversity of testing criteria as well their complementary aspects, some theoretical and empirical studies have been conducted, aiming at establishing an effective, low-cost testing strategy [5][14][12][20][21][15][22][3][17]. Effectiveness, cost, and strength are the three most meaningful bases against which test adequacy criteria can be compared. Effectiveness is related to the fault detection capability of a criterion; cost indicates the effort to satisfy a criterion; and strength refers to the difficulty of satisfying a given criterion $C_2$ for a test set $T$ that already satisfies another criterion $C_1$.

Mutation Testing, originally proposed by DeMillo *et al*. [9], although powerful, is computationally expensive [20][21][15]. Its high cost of application, mainly due to the high number of mutants created, has motivated the proposition of alternative criteria for its application [1][11][14][15]. One of these alternatives tries to reduce the cost of Mutation Testing application by determining a sufficient mutant operators set [15].

This work investigates approaches for the determination of a sufficient mutant operators set for C, aiming at contributing to the establishment of low-cost, effective mutation based testing strategies. We designed a procedure for the determination of a sufficient mutant operators set for C language, named Sufficient Procedure [3][4], based on guidelines we have established inspired in previous work.

We carried out two experiments: Experiment I, using a set of 27 programs, part of a text editor; and Experiment II, using 5-Unix utility programs. These experiments have been conducted using *Proteum* [6], an acronym for *PROgram TEsting Using Mutants*, a tool that supports the testing of C programs at the unit level.

The remainder of this paper is organized as follows. In Section 2, an overview of mutation testing is provided as well as the related work is described. In Section 3 we discuss the guidelines we thought about to establish the Sufficient Procedure. Sections 4 and 5 contain the description and analysis of the experiments we carried out. In Section 6 we compare the results of

Experiment I and Experiment II. In Section 7, our conclusions and further work are presented.

## 2 MUTATION TESTING: AN OVERVIEW

Mutation Testing is a fault-based testing criterion [9]. This criterion is based on the assumption that a program will be well tested if all so-called "simple faults" are detected and removed.

Simple faults are introduced into the program by creating different versions of the program, known as mutants, each of which containing a simple syntactic change. The simple faults are modeled by a set of mutant operators applied to a program *P* under testing.

The quality of test set *T* is measured by its ability to distinguish the behavior of the mutants from the behavior of the original program. So, the goal is to find a test case that causes a mutant to generate a different output from that of the original program. A mutant is considered equivalent if no such test case exists. If *P* behaves as per the specification when *T* is applied, then the quality of *T* is demonstrated; otherwise, a fault has been detected and the debugging activity would take place.

A test set that kills all non-equivalent mutants is said to be adequate relative to Mutation Testing, denoted by MT-adequate. The mutation score is the ratio of the number of dead mutants to the number of non-equivalent mutants; it measures the adequacy of test set. It should be observed that it is expected that complex faults be coupled to simple faults in such a way that a test set that detects all simple faults in a program will detect most complex faults. This is the so-called coupling effect assumption [9].

### 2.1 Alternative Mutation Testing Criteria

Some empirical studies have provided evidences that Mutation Testing is among the most promising criteria in terms of fault detection [20][21][15][22]. However, as highlighted before, Mutation Testing often imposes unacceptable demands on computing and human resources because of the large number of mutants that need to be compiled and executed on one or more test cases. In addition, a tester needs to examine many mutants and analyze them for possible equivalence with the program under testing. For these reasons, Mutation Testing is generally regarded as too expensive to use.

The test community, to deal with the cost aspects, has investigated some approaches derived from Mutation Testing: Randomly Selected Mutation [1], Constrained Mutation [11] and Selective Mutation [14][15]. In fact, the goal is to determine a set of mutations in such a way that if we obtain a test set *T*, which is able to distinguish those mutations, *T* will also be MT-adequate. In other words, the idea is that several mutants can lead to the same test case selection, so that we can use subsets of operators or mutants that lead to select test sets as effective as the total set of operators and mutants would [3][8].

Randomly Selected Mutation, proposed by Acree *et al*. [1], considers a percentage of the mutants generated by each operator (*x*%). Empirical studies conducted for FORTRAN and C programs [5][21] indicated that it is possible to obtain high mutation scores even with a reduced number of mutants. However, the randomly selection of mutants ignores the fault detection capability of individual mutant types [22]. Budd's fault detection experiments [5] found that mutants generated with respect to one mutant operator may be more effective in detecting certain types of faults than mutants generated with respect to another operator. This suggests that while mutants are selected for examination, they should be weighted differently depending on their respective fault detection capability.

Mathur proposed a variant of Acree *et al*.'s idea: Constrained Mutation [11]. In Constrained Mutation we select a subset of mutant operators to be used in mutant generation. Satisfactory results have been obtained [20][21]. It is important to observe, however, that Constrained Mutation does not establish a method for selecting the operators to be used; in general, these operators are intuitively selected based on the authors' experience. The definition of systematic ways for selecting the operators may lead us to better results.

Offutt *et al*. introduced Selective Mutation [14]. In this approach, the method for selecting the operators is related to the quantity of mutants that each operator generates: the operators that create the most mutants are not applied. So, the *N*-Selective Mutation omits the *N* most prevalent operators. In another study, in the same research line, Offutt *et al*. [15] introduced the concept of sufficient mutant operators. The idea is to determine a set of sufficient mutant operators *S* in a such way that obtained a test set *T S*-adequate *T* would lead to a very high mutation score. Next, two relevant studies related to the determination of sufficient mutant operators are described. In the remainder of this paper we refer to these approaches as Selective Mutation.

### 2.2 Related Work

Offutt *et al*. conducted an experiment for the determination of sufficient mutant operators for FORTRAN language [15], using the *Mothra* tool [10]. The 22 mutant operators implemented in this tool are divided into three mutation classes: replacement of operands, expression modification and statement modification. Offutt *et al*. compared the mutation classes pairwise and noticed that with the five operators of expression modification class it was possible to obtain a significant reduction in the number of mutants generated (77.56%), preserving a high mutation score with respect to Mutation Testing (above 0.980). One important point to be observed is that all the sufficient mutant operators determined were not among the most six prevalent FORTRAN operators, meaning that this approach would improve the 6-Selective criterion.

In another experiment, conducted by Wong *et al*. [22], the Selective Mutation was investigated in the context of C and FORTRAN. For C language, it was

used *Proteum* [6], a testing tool that allows measuring the adequacy of the test sets with respect to (w.r.t.) 71 mutant operators, categorized in four mutation classes [2]: statement (15), operator (46), variable (7) and constant (3). Six selective mutation categories were constructed, based on 11 of the 71 mutant operators. These mutant operators were selected based on the authors' judgement of their relative usefulness. According to the authors, 6 of the 11 operators may constitute a very good starting point for establishing a sufficient set of mutant operators to use in an alternate cost-effective mutation. Some of the *Proteum* mutant operators are illustrated in Table 1.

Table 1. Sample of *Proteum* Mutant Operators [2]

| Mutant Operator | Description |
|---|---|
| SMTC | n-trip continue |
| SSDL | statement deletion |
| STRP | trap on statement execution |
| SWDD | *while* replacement by *do-while* |
| OASN | arithmetic operator by shift operator |
| OEBA | plain assignment by bitwise assignment |
| OLBN | logical operator by bitwise operator |
| OLLN | logical operator mutation |
| OLNG | logical negation |
| ORRN | relational operator mutation |
| VTWD | twiddle mutations |
| VDTR | domain traps |
| Cccr | constant for constant replacement |
| Ccsr | constant for scalar replacement |
| CRCR | required constant replacement |

We reproduced the experiments conducted by Offutt *et al*. and Wong *et al*. for two other sets of programs: a suite of 27-C programs which composed a simplified text editor, previously used by Weyuker [18]; and 5-Unix utility programs previously used by Wong *et al*. [23]. Applying the strategy of Offutt *et al*. on the 27-program suite we obtained the constant mutation class as the sufficient set, with a mutation score of 0.97143 and a cost reduction of 78.115%, in terms of the number of generated mutants. For the 5-program suite we obtained the operator mutation class as the sufficient set, with a mutation score of 0.99042 and a cost reduction of 66.269%. Applying the operators investigated by Wong *et al*. [22] we obtained a mutation score of 0.97979 and a cost reduction of 79.738% for the 27-program suite and a mutation score of 0.99195 with a cost reduction of 83.435% for the 5-program suite [3].

The results obtained with the intuitively set of operators proposed by Wong *et al*. were little better than the results obtained with the sufficient set obtained with the application of Offutt *et al*.'s strategy in the context of C language for the two suites of programs. It should be highlighted that the sufficient operators determined by Offutt *et al*.'s approach for the 27-program suite were among the most prevalent ones for C-language, conflicting with *N*-Selective mutation. Moreover, the sufficient operators were completely different for each program suite. Motivated by these results and based on Offutt *et al*.'s idea, we defined the Sufficient Procedure [3][4], a systematic way to select a set of sufficient mutant operators, based on the guidelines discussed in the next section.

## 3 GUIDELINES FOR DETERMINATION OF A SUFFICIENT MUTANT OPERATORS SET

Consider $MC_1$, $MC_2$, ..., $MC_n$ sets that represent mutant operators classes. Mutation Testing (MT) uses, in its original conception, the set of all mutant operators defined by $OP = MC_1 \cup MC_2 \cup ... \cup MC_n$. Any subset of the mutant operators $SC \in 2^{\{OP\}}$ establishes a selective criterion.

Given a selective criterion *SC*, a test set *T* is said to be *SC*-adequate if *T* obtains a mutation score of 1.000 w.r.t. the mutants generated by the *SC* mutant operators, i.e., if *T* is able to reveal the behavioral differences among *P* (program under test) and the non-equivalent mutants created by the *SC* operators. From now on, if *SC* is composed by only one operator *op* (*SC* = {*op*}), it will be used simply *op*.

A meaningful mechanism used to compare the testing criteria is the Inclusion Relation, defined by Rapps and Weyuker [16]. Let $C_1$ and $C_2$ be testing criteria. $C_1$ includes $C_2$ ($C_1 \Rightarrow C_2$) if for every test set $T_1$ $C_1$-adequate, $T_1$ is also $C_2$-adequate and there is some $T_2$ $C_2$-adequate that is not $C_1$-adequate; $C_1$ and $C_2$ are equivalent if for any $T$ $C_1$-adequate, $T$ is $C_2$-adequate and vice-versa. Based on this, other relations have been defined: ProbBetter [19], related to the effectiveness of the criteria; and ProbSubsume [12], related to the strength. For instance, a testing criterion $C_1$ ProbSubsumes $C_2$ for a program *P* if a test set *T* that is adequate with respect to $C_1$ is "likely" to be adequate with respect to $C_2$. If $C_1$ ProbSubsumes $C_2$, $C_1$ is said to be at least as more difficult to satisfy than $C_2$.

The underlying concepts of these relations leaded us to define the empirically adequacy concept and the EmpSubsumes Relation. In practice, for time and cost constraints, obtaining a mutation score near to 1.000 may be satisfactory. Offutt *et al*. argument that the software testing literature offers no clear evidence that 100% coverage provides better testing than coverage at a lower level [15]. Let *ms** be a mutation score defined by the tester. For a criterion *C* and a test set *T*, *T* is said to be empirically adequate to *C* (denoted by *T* is *C*-adequate*) if *T* obtains a mutation score equal or greater than *ms** w.r.t. *C* [3]. We assume that $C_1$ EmpSubsumes $C_2$ with a mutation score *ms** (denoted by $C_1 \overset{ms^*}{\Rightarrow} C_2$) if for every test set $T_1$ $C_1$-adequate, $T_1$ is also $C_2$-adequate* and there is some $T_2$ $C_2$-adequate that is not $C_1$-adequate*. $C_1$ and $C_2$ are empirically equivalent (denoted by equivalent*) if for any *T* $C_1$-adequate, *T* is also $C_2$-adequate* and vice-versa.

The determination of a sufficient mutant operators set consists in selecting a subset $SS \in 2^{\{OP\}}$, where *OP* is the total set of mutant operators defined for a target language, such that if a test set *T* is *SS*-adequate, *T* will also be *OP*-adequate*. In other words, if *T* obtains a mutation score of 1.000 w.r.t. *SS*, *T* will have a mutation score equal or greater than *ms** w.r.t. *OP* (Mutation Testing). It is important to observe that, given a mutation score *ms**, there is not only one sufficient mutant operator set.

Given the testing criteria $C_1$ and $C_2$, $C_1$ determines a high mutation score w.r.t. $C_2$ if every test set $T$ $C_1$-adequate has a high mutation score w.r.t. $C_2$, i.e., if $T$ is able to distinguish the most mutants generated by the mutant operators of $C_2$.

Since *SS* should determine a high mutation score w.r.t. *OP*, we establish some guidelines to be considered in selecting the mutant operators that will compose *SS* [3]. These guidelines lead to consider information on subsumption of some mutant operators by others, as motivated by Offutt *et al*. [15].

i. **Consider mutant operators that determine a high mutation score**
To guarantee that the sufficient mutant operators set determines a high mutation score w.r.t. Mutation Testing, we should select the operators that determine the greatest mutation scores w.r.t. the total set of mutant operators (*OP*). In same aspect, this capture the mutant operator effectiveness in the sense used by Offutt *et al*. [15], i.e., its mutation score against the full set of operators.

ii. **Consider one operator of each mutation class**
Each mutation class models specific errors in certain elements of a program (e.g. statements, operators, variables and constants). So, it is desirable that the sufficient set has, at least, the most representative operator of each class.

iii. **Evaluate the empirical inclusion among the mutant operators**
The mutant operators that are empirically included by other mutant operators of the sufficient set should be removed since these operators increase the application cost of the sufficient set, in terms of number of mutants and equivalence determination, and do not effectively contribute to the improvement of the testing activity.

iv. **Establish an incremental strategy**
Given the application cost and the test requirements that each mutation class determines, it is interesting to establish an incremental strategy of application among the mutant operators of the sufficient set. The idea is to apply, at first, the mutant operators that are relevant to certain minimal requirements of testing (e.g., all-nodes and all-edges coverage). Next, depending on the criticality of the application and the budget and time constraints, the mutant operators related to other concepts and test requirements may be applied.

v. **Consider mutant operators that provide an increment in the mutation score**
In general, independently of the quality of the test set, 80% of the mutants are killed at the first execution [5]. Considering that just around 20% of the mutants effectively contribute to the quality improvement of the test set, an increment of 1% in the mutation score represents 5% of the mutants that are really significant. Thus, the non-selected operators that if included in the sufficient set would increase the mutation score should be analyzed.

vi. **Consider mutant operators with high strength**
Other operators that should be considered to determine the sufficient set are those that have a high average strength w.r.t. each operator of the total set of operators.

The proposed Sufficient Procedure [3][4] has been refined through the conduction of two experiments, discussed in the next sections. It has six steps, related to the guidelines discussed above:

Step 1: Select mutant operators that determine a high mutation score w.r.t. *OP*.

Step 2: Select one operator of each mutation class.

Step 3: Reduce the preliminary sufficient set ($SS_{prel}$).

Step 4: Establish an incremental strategy.

Step 5: Select mutant operators that provide an increment in the mutation score.

Step 6: Select mutant operators with high strength w.r.t. *OP*.

## 4 EXPERIMENT I

The methodology used to conduct this experiment comprises five phases: Program Selection, Tool Selection, Test Set Generation, Sufficient Procedure Application and Data Analysis.

### 4.1 Program Selection

A suite of 27 small programs, part of a simplified text editor, was selected. These programs, originally written in Pascal, were converted to C; Weyuker [18] has also used them. As illustrated in Table 2, these programs range in size from 11 up to 71 executable statements and have 119 up to 1631 mutants. In Table 2 it is also provided information on the number of mutants per mutation class as well as on the number of equivalent mutants.

Table 2. Program Suite I:
Number of LOC and Mutants

| Program | LOC | Mutants Total / Equiv | Statement Total / Equiv | Operator Total / Equiv | Variable Total / Equiv | Constant Total / Equiv |
|---|---|---|---|---|---|---|
| *append* | 13 | 387 / 64 | 54 / 5 | 183 / 34 | 80 / 19 | 70 / 6 |
| *archive* | 14 | 514 / 75 | 57 / 0 | 222 / 19 | 61 / 28 | 174 / 28 |
| *change* | 13 | 119 / 19 | 59 / 5 | 15 / 8 | 33 / 3 | 12 / 3 |
| *ckglob* | 23 | 730 / 234 | 76 / 15 | 340 / 99 | 172 / 78 | 142 / 42 |
| *cmp* | 13 | 430 / 41 | 78 / 1 | 129 / 6 | 119 / 34 | 104 / 0 |
| *command* | 71 | 1207 / 243 | 230 / 26 | 248 / 89 | 345 / 86 | 384 / 42 |
| *compare* | 18 | 482 / 43 | 74 / 2 | 157 / 21 | 153 / 16 | 98 / 4 |
| *compress* | 14 | 454 / 87 | 69 / 0 | 215 / 53 | 110 / 34 | 60 / 0 |
| *dodash* | 15 | 1071 / 202 | 62 / 1 | 423 / 71 | 341 / 91 | 245 / 39 |
| *edit* | 23 | 524 / 139 | 105 / 13 | 211 / 63 | 126 / 45 | 82 / 18 |
| *entab* | 18 | 370 / 39 | 89 / 2 | 144 / 20 | 79 / 17 | 58 / 0 |
| *expand* | 15 | 389 / 37 | 67 / 0 | 210 / 20 | 60 / 17 | 52 / 0 |
| *getcmd* | 32 | 860 / 19 | 574 / 1 | 44 / 14 | 10 / 4 | 232 / 0 |
| *getdef* | 31 | 840 / 134 | 136 / 7 | 308 / 54 | 208 / 47 | 188 / 26 |
| *getfn* | 11 | 494 / 88 | 50 / 0 | 245 / 48 | 90 / 26 | 109 / 14 |
| *getfns* | 23 | 627 / 135 | 64 / 5 | 233 / 54 | 162 / 44 | 168 / 32 |
| *getlist* | 21 | 678 / 93 | 86 / 4 | 310 / 56 | 150 / 30 | 132 / 3 |
| *getnum* | 17 | 564 / 73 | 71 / 2 | 252 / 19 | 107 / 31 | 134 / 21 |
| *getone* | 23 | 834 / 122 | 98 / 4 | 397 / 67 | 205 / 38 | 134 / 13 |
| *gtext* | 16 | 804 / 82 | 55 / 3 | 331 / 37 | 224 / 39 | 194 / 3 |
| *makepat* | 29 | 1683 / 266 | 151 / 4 | 516 / 98 | 594 / 108 | 422 / 56 |
| *omatch* | 36 | 840 / 220 | 146 / 18 | 313 / 100 | 196 / 57 | 185 / 45 |
| *optpat* | 13 | 444 / 138 | 68 / 7 | 218 / 75 | 70 / 23 | 88 / 33 |
| *spread* | 19 | 1061 / 79 | 86 / 0 | 418 / 40 | 355 / 35 | 202 / 4 |
| *subst* | 36 | 1631 / 300 | 154 / 9 | 497 / 105 | 739 / 149 | 241 / 37 |
| *translit* | 33 | 1125 / 121 | 140 / 5 | 447 / 73 | 301 / 37 | 237 / 6 |
| *unrotate* | 28 | 984 / 43 | 92 / 0 | 441 / 20 | 189 / 19 | 262 / 4 |
| **Total** | **618** | **20146 / 3136** | **2991 / 139** | **7467 / 1363** | **5279 / 1155** | **4409 / 479** |

### 4.2 Tool Selection

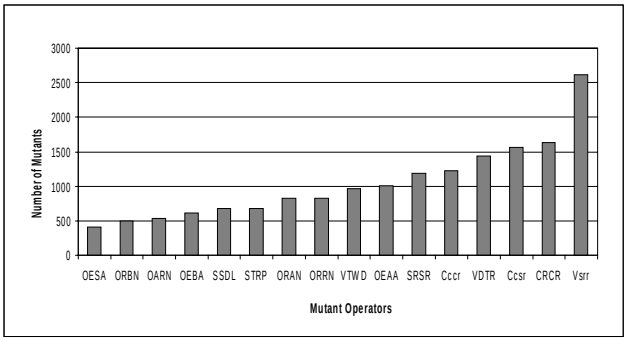
We used *Proteum* testing tool [6], developed at University of São Paulo, which supports Mutation Testing application to C programs.

### 4.3 Test Set Generation

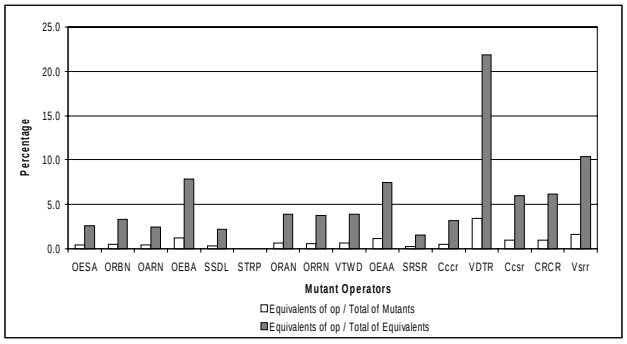
One *ad hoc* test set was generated for each program based on its specification, i.e., none functional criterion has been used. Next, these test sets were improved based on their adequacy w.r.t. Mutation Testing: new test cases were added to obtain a MT-adequate test set for each one of the 27 programs. We kept in these sets only effective test cases, i.e., test cases that killed at least one mutant.

### 4.4 Sufficient Procedure Application

For each program, the cost of the mutant operators was determined. In Graphic 1 we provide information on the mutant operators cost in terms of number of generated mutants and in terms of number of equivalent mutants for the most prevalent ones. From 71, only 39 operators were applicable (i.e., generate at least one mutant) to the 27-program suite: 10 of statement, 21 of operator, 5 of variable and 3 of constant.




(a)




(b)

Graphic 1. Program Suite I:
Mutant Operators Cost per Number of:
(a) Generated Mutants and (b) Equivalent Mutants

Following, the mutation score of the adequate test sets for each operator $op_i$ w.r.t. each other operator $op_j$ of the total set of operators was determined, i.e., the capability of a $op_i$-adequate test set to distinguish the mutants of each operator $op_j \in OP$.

In Table 3 a sample of the average mutation score per operator obtained for the 27-program suite is presented.

Table 3. Program Suite I:
Sample of the Average Mutation Score per Operator

| op \ op | Cccr | Ccsr | CRCR | ORRN | SSDL | SSWM | VTWD | ... | Average |
|---|---|---|---|---|---|---|---|---|---|
| Cccr | 1.000 | 0.887 | 0.904 | 0.921 | 0.936 | 0.865 | 0.923 | ... | 0.922 |
| Ccsr | 0.902 | 1.000 | 0.992 | 0.938 | 0.908 | 0.835 | 0.974 | ... | 0.948 |
| CRCR | 0.901 | 0.982 | 1.000 | 0.915 | 0.910 | 0.835 | 0.973 | ... | 0.937 |
| ORRN | 0.900 | 0.863 | 0.870 | 1.000 | 0.921 | 0.465 | 0.865 | ... | 0.897 |
| SSDL | 0.909 | 0.853 | 0.881 | 0.887 | 1.000 | 1.000 | 0.868 | ... | 0.885 |
| SSWM | 0.470 | 0.580 | 0.500 | 0.890 | 0.840 | 1.000 | 0.495 | ... | 0.594 |
| VTWD | 0.902 | 0.950 | 0.969 | 0.916 | 0.905 | 0.800 | 1.000 | ... | 0.928 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| Average | 0.732 | 0.719 | 0.730 | 0.767 | 0.786 | 0.544 | 0.730 | ... | – |

From Table 3 we can extract the following information:

- The average mutation score of $op_i$-adequate test sets w.r.t. all $op_j \in OP$. For instance, on average, the mutation score that ORRN (fifth line) determines w.r.t. Ccsr (third column) is 0.863, i.e., the ORRN-adequate test sets are able to distinguish 86.3% of the mutants of Ccsr.
- The average mutation score of $op_i$-adequate test sets w.r.t. *OP*. For instance, on average, ORRN-adequate test sets determine a mutation score of 0.897 w.r.t. Mutation Testing.
- The average strength of $op_i$ w.r.t. $op_j$, $op_i \in OP$. For instance, on average, the strength of Ccsr w.r.t. ORRN is 0.137 (1 - 0.863).
- The average strength of $op_i$ w.r.t. *OP*. For instance, on average, the strength of Ccsr w.r.t. Mutation Testing is 0.281 (1 - 0.719).

From Graphic 1(a) and Table 3, we obtained Table 4. This table presents the mutant operators ordered according the mutation score, strength and cost – required information for applying the Sufficient Procedure.

Table 4. Program Suite I:
Order of Operators According to:
(a) Mutation Score, (b) Strength and (c) Cost

| (a) | (b) | (c) |
|---|---|---|
| **Mutation Score** | **Strength** | **Cost** |
| Ccsr (0.948) | SSWM (0.456) | Vsrr (2620) |
| Vsrr (0.948) | SWDD (0.364) | CRCR (1631) |
| CRCR (0.937) | OABN (0.353) | Ccsr (1559) |
| VTWD (0.928) | SMTC (0.339) | VDTR (1437) |
| Cccr (0.922) | OLSN (0.338) | Cccr (1219) |
| ORRN (0.897) | Vprr (0.323) | SRSR (1193) |
| VDTR (0.891) | OASN (0.310) | OEAA (1010) |
| SSDL (0.885) | OLLN (0.291) | VTWD (958) |
| ORSN (0.873) | OARN (0.283) | ORAN (830) |
| ORAN (0.872) | Ccsr (0.281) | ORRN (830) |
| SRSR (0.866) | CRCR (0.270) | STRP (677) |
| ORBN (0.856) | VTWD (0.270) | SSDL (676) |
| ... | ... | ... |

To apply the Sufficient Procedure we used $ms^* = 0.99$, in fact the same index considered in Offutt *et al.*'s study [15]. Table 7 contains the preliminary sufficient mutant operators sets ($SS_{prel}$) obtained at each step of the procedure application. The set of mutant operators that are not present in $SS_{prel}$ is referred to as $\overline{SS}_{prel}$, i.e., $\overline{SS}_{prel} = OP - SS_{prel}$.

In Step 1, considering *AIMS* (Average Index of Mutation Score) = 0.900 ± 0.005 and Table 4(a), we obtained $SS_{prel}$ = {Ccsr, Vsrr, CRCR, VTWD, Cccr, ORRN}.

In Step 2 the operator SSDL was included in $SS_{prel}$, since it did not contain any operator of the statement mutation class. Moreover, SSDL, according to Table 4(a), on average, determines the greatest mutation score w.r.t. the total set of operators among the statement mutation class and $SS_{prel} \overset{0.99}{\not\Rightarrow} \{SSDL\}$.

In Step 3, as CRCR was the most empirically included among the operators of $SS_{prel}$, it was removed. Repeating this step the operator Vsrr was also removed.

In Step 4, considering Table 4(c), we determined SSDL, ORRN, VTWD, Cccr e Ccsr as the incremental order to apply the mutant operators of $SS_{prel}$.

In Step 5, in this case, we aimed at including in $SS_{prel}$ at most one additional operator of each mutation class. We also defined *IMI* (Index of Minimum Increment) = 0.001, what represents to add to $SS_{prel}$ those operators that allow distinguishing at least 0.5% of the significant mutants. According to Table 5, which contains information about the mutation score increment each operator of $\overline{SS}_{prel}$ provides in the first iteration, the nine first operators were considered. Starting with the statement mutation class, we observed that SMTC (0.003875) was the operator that determines the greatest increment; in the same range of increment (0.003) was also the operator SMTT (0.003386). As $SS_{prel}$ empirically included neither SMTC nor SMTT and since the strength of SMTC was the greatest, SMTC was added to $SS_{prel}$. This step was repeated looking for the operator, variable and constant mutations. At the end, SMTC, OLBN and VDTR were included in $SS_{prel}$ and an increment of 0.007531 was obtained, what represents more than 3.7% of the significant mutants.

Table 5. Program Suite I: Mutation Score Increment

| Mutant Operator | Index of Increment | $SS_{prel}$ Score + Increment |
|---|---|---|
| SMTC | 0.003875 | 0.992269 |
| Varr | 0.003407 | 0.991801 |
| SMTT | 0.003386 | 0.991780 |
| VDTR | 0.002425 | 0.990819 |
| Vsrr | 0.001525 | 0.989918 |
| OLAN | 0.001286 | 0.989680 |
| OLRN | 0.001273 | 0.989667 |
| OLBN | 0.001231 | 0.989624 |
| ORAN | 0.001116 | 0.989510 |
| OEAA | 0.000869 | 0.989263 |
| ORBN | 0.000803 | 0.989197 |
| ... | ... | ... |
| OABN | 0.000118 | 0.988512 |
| Vprr | 0.000026 | 0.988420 |

In Step 6, considering Table 4(b) and *AIS* (Average Index of Strenght) = 0.300 ± 0.005, the operators SSWM, SWDD, OABN, SMTC, OLSN, Vprr and OASN were taken of high strength. According to Table 6, which contains the mutation score that $SS_{prel}$ determines w.r.t. the operators of high strength, SWDD and OASN were not empirically included by $SS_{prel}$ and were taken in consideration. Since SWDD presented the greatest strength, it was added to $SS_{prel}$. Repeating this step the operator OASN was also included. At the end of Step 6, the final sufficient mutant operators set was *SS*-27 = {SWDD, SMTC, SSDL, OLBN, OASN, ORRN, VTWD, VDTR, Cccr, Ccsr}. It is important to observe that the greater the *ms** value is the greater can be the number of mutant operators included. For instance, if we had used *ms** = 0.98 the operator OASN would not have been included in *SS*-27.

Table 6. Program Suite I:
$SS_{prel}$: Score of the High Strength Operators

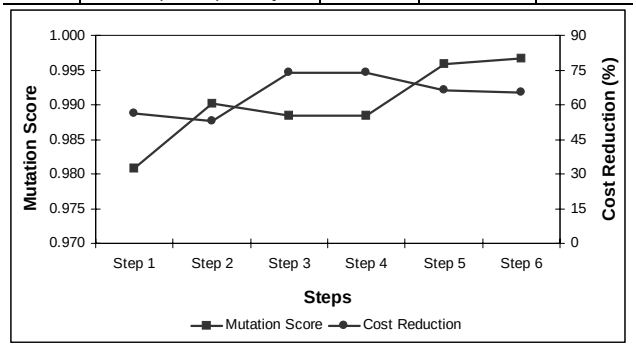
| Mutant Operator | Mutation Score |
|---|---|
| SWDD | 0.893 |
| OASN | 0.982 |
| Vprr | 0.993 |
| OABN | 0.996 |
| OLSN | 1.000 |
| SSWM | 1.000 |

### 4.5 Data Analysis

In this section we carry out some analysis with the data obtained with the procedure application. The preliminary sufficient sets ($SS_{prel}$) and the final sufficient set (*SS*-27) are analyzed w.r.t. Mutation Testing. The mutation score (MS), the cost reduction (CR) and the cost/benefit (CR/MS) evolution are presented in Table 7 and in Graphic 2. Notice that for the relation CR/MS we are only interested in sets with a high mutation score, otherwise a set with a cost reduction of 99% and a mutation score of 0.01 would also look good.

Table 7. Program Suite I: Results Obtained at Each Step of the Sufficient Procedure

| Step | Mutant Operators | MS | CR (%) | CR/MS |
|---|---|---|---|---|
| 1 | {Ccsr, Vsrr, CRCR, VTWD, Cccr, ORRN} | 0.98087 | 56.234 | 0.573 |
| 2 | {Ccsr, Vsrr, CRCR, VTWD, Cccr, ORRN, SSDL} | 0.99014 | 52.879 | 0.534 |
| 3 | {Ccsr, VTWD, Cccr, ORRN, SSDL} | 0.98839 | 73.980 | 0.748 |
| 4 | {SSDL, ORRN, VTWD, Cccr, Ccsr} | 0.98839 | 73.980 | 0.748 |
| 5 | {SMTC, SSDL, OLBN, ORRN, VTWD, VDTR, Cccr, Ccsr} | 0.99592 | 65.988 | 0.663 |
| 6 | {SWDD, SMTC, SSDL, OLBN,OASN, ORRN, VTWD, VDTR, Cccr, Ccsr} | 0.99660 | 65.015 | 0.652 |

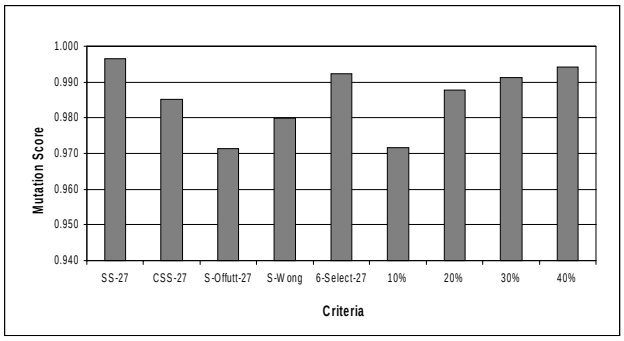



Graphic 2. Program Suite I: Sufficient Set Evolution

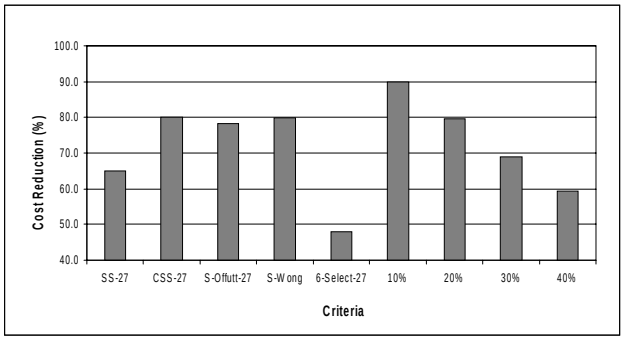
We also carry out a comparison among the sets: *SS*-27; *CSS*-27 (the constrained sufficient set from *SS*-27, composed of the most representative operator of each mutation class, i.e., the operator that determines the greatest mutation score for that class); *S-Offutt*-27 (the operators obtained from Offutt *et al*.'s strategy); *S-Wong* (the operators proposed by Wong *et al*.); 6-*Selective*-27 mutation; and some randomly mutation criteria (10%, 20%, 30% and 40%). The results in terms of mutation score (MS), cost reduction (CR) and cost/benefit (CR/MS) are summarized in Table 8 and in Graphic 3.

Table 8. Program Suite I: Comparison among Selective and Randomly Mutation

| Criterion | MS | CR (%) | CR/MS |
|---|---|---|---|
| *SS*-27 | 0.99660 | 65.015 | 0.652 |
| *CSS*-27 | 0.98505 | 80.031 | 0.812 |
| *S-Offutt*-27 | 0.97143 | 78.115 | 0.804 |
| *S-Wong* | 0.97979 | 79.738 | 0.814 |
| 6-*Selective*-27 | 0.99242 | 47.945 | 0.483 |
| 10%-*Randomly* | 0.97160 | 90.038 | 0.926 |
| 20%-*Randomly* | 0.98791 | 79.554 | 0.805 |
| 30%-*Randomly* | 0.99120 | 69.006 | 0.696 |
| 40%-*Randomly* | 0.99420 | 59.352 | 0.597 |




(a)




(b)

Graphic 3. Program Suite I: Selective and Randomly Mutation: (a) Mutation Score and (b) Cost Reduction

On average we can observe that:

- The *SS*-27 criterion determines the greatest mutation score followed by 40%-*Randomly*, although the cost reduction they provide are not the greatest.
- The 10%-*Randomly* criterion provides the greatest cost reduction followed by *CSS*-27.
- In terms of cost/benefit the best criterion is 10%-*Randomly* followed by *S-Wong* and *CSS*-27.

Another relevant aspect to be considered is the effectiveness of the criteria. If we consider effectiveness by computing the mutation score of the adequate test sets w.r.t. a specific criterion against the full set of operators, we would recommend *SS*-27, 6-*Selective*-27 and 40%-*Randomly*, respectively, as they determine mutation scores greater than 0.990 and are in the same range of cost/benefit. Other approach would be measuring the relative abilities of the adequate test sets w.r.t a criterion *C* to detect actual faults in the programs. Wong *et al*. [20][21][22] have provided evidences that Selective Mutation is more effective in revealing faults than Randomly Mutation. Giving these considerations, we focus our analysis on the selective criteria.

Table 9 provides a strength analysis among the selective criteria. For example, the strength of *SS*-27 w.r.t. *S-Offutt*-27 is measured by calculating the average mutation score that *S-Offutt*-27-adequate test sets provide w.r.t. the mutants generated by the operators of *SS*-27. We can observe that:

- For all programs, the mutation score provided by *SS*-27 is equal or greater than those obtained with *S-Offutt*-27 and *S-Wong*.
- *SS*-27 includes *S-Wong* and *CSS*-27, i.e., *SS*-27-adequate test sets are able to distinguish all mutants generated by the operators of *S-Wong* and *CSS*-27; the inverse is not true.
- *SS*-27 and *S-Offutt*-27, and *SS*-27 and 6-*Selective*-27 are incomparable in the perspective of the inclusion relation. However, *SS*-27 empirically includes *S-Offutt*-27 and 6-*Selective*-27, i.e., *SS*-27-adequate test sets are able to distinguish, on average, 99.9% and 99.6%, respectively, of the mutants generated by the *S-Offutt*-27 and 6-*Selective*-27 operators while *S-Offutt*-27-adequate test sets and 6-*Selective*-27-adequate test sets are able to distinguish 97.2% and 98.8%, respectively, of the mutants generated by the operators of *SS*-27.

Table 9. Program Suite I: Strength Analysis

| Criteria | Mutation Score |
|---|---|
| *SS*-27 × *CSS*-27 | 1.00000 |
| *CSS*-27 × *SS*-27 | 0.98353 |
| *SS*-27 × *S-Offutt*-27 | 0.99902 |
| *S-Offutt*-27 × *SS*-27 | 0.97157 |
| *SS*-27 × *S-Wong* | 1.00000 |
| *S-Wong* × *SS*-27 | 0.98353 |
| *SS*-27 × 6-*Selective*-27 | 0.99650 |
| 6-*Selective*-27 × *SS*-27 | 0.98798 |

The uniformity of the mutation scores determined by the selective criteria for all the programs, illustrated in Table 10(a), is also a relevant data. Table 10(b) provides the corresponding cost reduction for each program. According to these tables, we notice that:

- *SS*-27 determines a mutation score equal 1.000 for 13 programs followed by 6-*Selective*-27 and *S-Wong* with 11 and 5 programs, respectively.
- *SS*-27 determines a mutation score greater than 0.990 for 25 programs followed by 6-*Selective*-27, *CSS*-27 and *S-Wong* with 20, 15 and 11 programs, respectively.

Table 10. Program Suite I: Distribution of (a) Mutation Score and (b) Cost Reduction

(a)

| Program | Mutation Score | | | | |
|---|---|---|---|---|---|
| | *SS*-27 | *CSS*-27 | *S-Offutt*-27 | *S-Wong* | 6-*Selective*-27 |
| *append* | 1.00000 | 0.99381 | 0.99071 | 1.00000 | 0.99381 |
| *archive* | 0.99772 | 0.98178 | 0.95900 | 0.98178 | 1.00000 |
| *change* | 1.00000 | 0.90000 | 0.76000 | 0.90000 | 1.00000 |
| *ckglob* | 1.00000 | 1.00000 | 0.98387 | 1.00000 | 1.00000 |
| *cmp* | 1.00000 | 1.00000 | 0.99743 | 0.98972 | 0.98972 |
| *command* | 0.99896 | 0.94606 | 0.96473 | 0.94917 | 0.93361 |
| *compare* | 1.00000 | 0.99089 | 0.95900 | 0.99089 | 1.00000 |
| *compress* | 1.00000 | 0.98638 | 0.98365 | 0.98638 | 1.00000 |
| *dodash* | 0.99540 | 0.99540 | 0.97814 | 0.98044 | 0.99425 |
| *edit* | 0.95844 | 0.94286 | 0.95584 | 0.95584 | 0.97922 |
| *entab* | 1.00000 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| *expand* | 0.99432 | 0.98579 | 0.96875 | 0.96591 | 1.00000 |
| *getcmd* | 1.00000 | 1.00000 | 1.00000 | 1.00000 | 1.00000 |
| *getdef* | 0.99858 | 0.99858 | 0.98725 | 0.99858 | 1.00000 |
| *getfn* | 1.00000 | 0.98522 | 0.97044 | 0.99015 | 0.99015 |
| *getfns* | 1.00000 | 0.99797 | 0.97967 | 0.98984 | 1.00000 |
| *getlist* | 0.99829 | 0.99316 | 0.99145 | 0.98803 | 0.98462 |
| *getnum* | 0.98982 | 0.98982 | 0.97149 | 0.96945 | 0.99389 |
| *getone* | 0.99438 | 0.98876 | 0.97472 | 0.99017 | 0.99579 |
| *gtext* | 1.00000 | 0.99862 | 1.00000 | 0.99862 | 0.99862 |
| *makepat* | 0.99788 | 0.99647 | 0.99788 | 0.98024 | 0.97459 |
| *omatch* | 1.00000 | 0.98548 | 0.97742 | 0.97742 | 0.98871 |
| *optpat* | 0.99346 | 0.97386 | 0.95098 | 0.93464 | 0.99673 |
| *spread* | 0.99796 | 0.98574 | 0.98065 | 0.99694 | 0.99491 |
| *subst* | 0.99700 | 0.99325 | 0.99025 | 0.98800 | 0.98875 |
| *translit* | 0.99602 | 0.99602 | 0.95518 | 0.95219 | 0.99801 |
| *unrotate* | 1.00000 | 0.99044 | 1.00000 | 1.00000 | 1.00000 |
| **Average** | **0.99660** | **0.98505** | **0.97143** | **0.97979** | **0.99242** |

(b)

| Program | Cost Reduction (%) | | | | |
|---|---|---|---|---|---|
| | *SS*-27 | *CSS*-27 | *S-Offutt*-27 | *S-Wong* | 6-*Selective*-27 |
| *append* | 68.217 | 82.429 | 81.912 | 79.845 | 38.501 |
| *archive* | 53.113 | 73.541 | 66.148 | 77.432 | 42.023 |
| *change* | 73.950 | 85.714 | 89.916 | 85.714 | 21.849 |
| *ckglob* | 64.110 | 78.356 | 80.548 | 74.658 | 39.178 |
| *cmp* | 60.233 | 75.116 | 75.814 | 67.907 | 52.093 |
| *command* | 54.598 | 73.985 | 68.186 | 79.619 | 60.646 |
| *compare* | 67.635 | 82.158 | 79.668 | 80.290 | 45.228 |
| *compress* | 66.960 | 78.855 | 86.784 | 70.925 | 35.463 |
| *dodash* | 64.146 | 79.739 | 77.124 | 76.004 | 45.285 |
| *edit* | 69.847 | 81.298 | 84.351 | 81.298 | 41.985 |
| *entab* | 65.135 | 77.297 | 84.324 | 74.865 | 38.378 |
| *expand* | 68.123 | 82.005 | 86.632 | 77.378 | 28.792 |
| *getcmd* | 69.186 | 95.465 | 73.023 | 95.000 | 83.488 |
| *getdef* | 64.762 | 80.595 | 77.619 | 82.857 | 47.024 |
| *getfn* | 62.753 | 77.530 | 77.935 | 76.316 | 36.437 |
| *getfns* | 59.330 | 75.279 | 73.206 | 80.383 | 50.080 |
| *getlist* | 65.634 | 77.729 | 80.531 | 76.696 | 39.676 |
| *getnum* | 61.879 | 76.241 | 76.241 | 74.468 | 36.702 |
| *getone* | 68.585 | 80.695 | 83.933 | 77.338 | 36.930 |
| *gtext* | 64.179 | 79.478 | 75.871 | 82.711 | 49.005 |
| *makepat* | 63.815 | 79.144 | 74.926 | 80.095 | 57.992 |
| *omatch* | 64.405 | 77.976 | 77.976 | 75.595 | 44.762 |
| *optpat* | 65.766 | 79.730 | 80.180 | 75.225 | 35.586 |
| *spread* | 68.992 | 81.904 | 80.961 | 83.223 | 50.236 |
| *subst* | 74.801 | 84.672 | 85.224 | 83.446 | 57.817 |
| *translit* | 64.711 | 78.578 | 78.933 | 79.111 | 43.644 |
| *unrotate* | 61.789 | 81.301 | 73.374 | 84.858 | 43.293 |
| **Average** | **65.015** | **80.031** | **78.115** | **79.738** | **47.945** |

- *SS*-27 determines a mutation score below 0.990 just for the program *getnum* (0.98982) and *edit* (0.95844).
- *SS*-27 does not determine a mutation score below 0.950 for any program. The same does not happen with *S-Offutt*-27, *S-Wong* and 6-*Selective*-27.
- *S-Offutt*-27, *CSS*-27 and *S-Wong* determine mutation scores of 0.760, 0.900 and 0.900, respectively, for the program *change*, while *SS*-27 and 6-*Selective*-27 present mutation scores of 1.000.
- The greatest cost reduction obtained with *SS*-27, *CSS*-27, *S-Offut*-27, *S-Wong* and 6-*Selective*-27 sets are 74.8%, 95.5%, 89.9%, 95.0% and 83.5% with a mutation score of 0.997, 1.000, 0.760, 1.000 and 1.000, respectively.
- The least cost reduction obtained with *SS*-27, *CSS*-27, *S-Offut*-27, *S-Wong* and 6-*Selective*-27 sets are 53.1%, 73.5%, 66.1%, 67.9% and 21.85% with a mutation score of 0.998, 0.982, 0.959, 0.988 and 1.000, respectively.

Considering mutation score, cost reduction, strength and mutation score distribution, the *SS*-27 set would constitute the best choice as it determines the greatest mutation score, empirically includes the other selective criteria and presents an excellent mutation score uniformity, although it does not present the best cost reduction. *CSS*-27 and *S-Wong* would also constitute a good choice as these criteria present the best cost/benefit relation. Another point that favors these sets is that the cost to obtain them is very low; in fact, for *S-Wong* there are no costs at all.

## 5 EXPERIMENT II

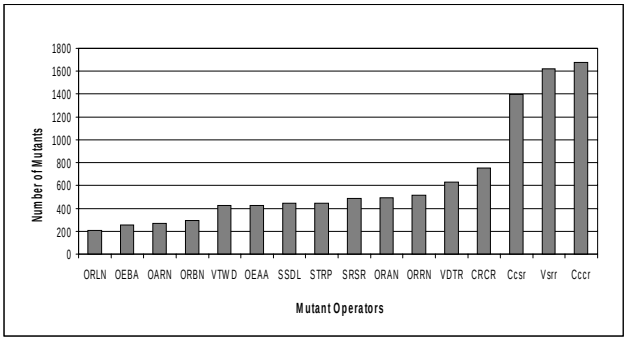
The same phases of Experiment I were also applied for a set of 5-Unix utility programs. The main difference was that, in Test Set Generation, 11 MT-adequate test sets were used for each one of the 5 programs. Initially we generated a pool of test cases composed by: 1) *ad hoc* functional test cases, based on program specification; and 2) randomly generated test cases. From this pool, 11 test sets were generated for each program. Next, we ran the test cases of each test set against the mutants and, if necessary, we added manual test cases to the set until we have obtained a MT-adequate test set.

As illustrated in Table 11, these programs range in size from 76 up to 119 executable statements and have 1619 up to 4332 mutants. In Table 11 it is also provided information on the number of mutants per mutation class as well as on the number of equivalent mutants.

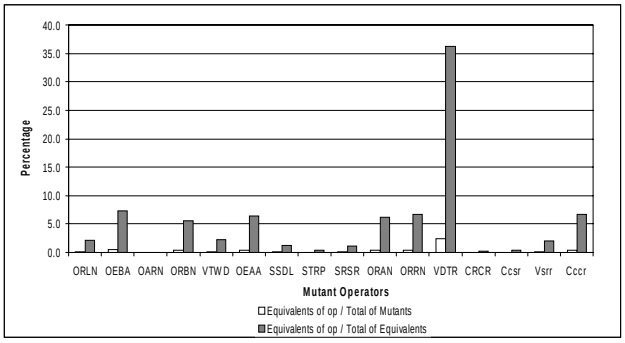
Table 11. Program Suite II: Number of LOC and Mutants

| Program | LOC | Mutants Total / Equiv | Statement Total / Equiv | Operator Total / Equiv | Variable Total / Equiv | Constant Total / Equiv |
|---|---|---|---|---|---|---|
| *cal* | 119 | 4332 / 221 | 352 / 3 | 1409 / 86 | 791 / 117 | 1780 / 15 |
| *checkeq* | 76 | 3099 / 206 | 268 / 1 | 937 / 99 | 783 / 106 | 1111 / 0 |
| *comm* | 119 | 1728 / 166 | 405 / 5 | 642 / 111 | 367 / 35 | 314 / 15 |
| *look* | 107 | 2056 / 143 | 319 / 23 | 720 / 36 | 646 / 55 | 371 / 29 |
| *uniq* | 103 | 1619 / 93 | 348 / 0 | 621 / 64 | 406 / 28 | 244 / 1 |
| **Total** | **524** | **12834 / 829** | **1692 / 32** | **4329 / 396** | **2993 / 341** | **3820 / 60** |

In Graphic 4 we provide information on the mutant operators cost in terms of number of generated mutants and in terms of number of equivalent mutants for the 5-program suite, for the most prevalent operators. 56 operators were applicable to the 5 programs: 14 of statement, 33 of operator, 6 of variable and 3 of constant.




(a)



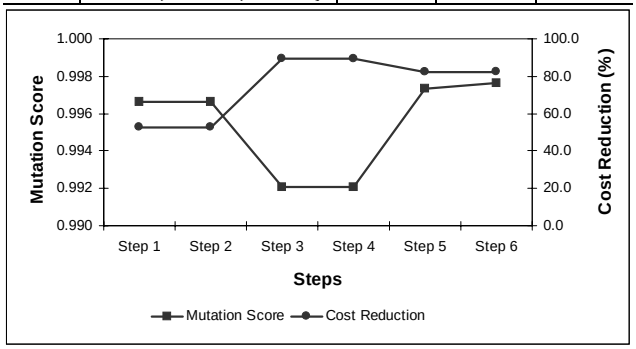

(b)

Graphic 4. Program Suite II:
Mutant Operators Cost per Number of:
(a) Generated Mutants and (b) Equivalent Mutants

For Experiment II, the sufficient mutant operators set was *SS*-5 = {SMTC, SSDL, OEBA, ORRN, VTWD, VDTR}. We carry out the same analysis as for the Experiment I. The preliminary sufficient sets ($SS_{prel}$) and the final sufficient set (*SS*-5) are analyzed w.r.t. Mutation Testing. The mutation score (MS), the cost reduction (CR) and the cost/benefit (CR/MS) evolution are presented in. Table 12 and in Graphic 5.

Table 12. Program Suite II: Results Obtained at Each Step of the Sufficient Procedure

| Step | Mutant Operators | MS | CR (%) | CR/MS |
|---|---|---|---|---|
| 1 | {Cccr, Vsrr, VTWD, SSDL, ORRN, Ccsr} | 0.99662 | 52.680 | 0.529 |
| 2 | {Cccr, Vsrr, VTWD, SSDL, ORRN, Ccsr} | 0.99662 | 52.680 | 0.529 |
| 3 | {VTWD, SSDL, ORRN} | 0.99209 | 89.224 | 0.899 |
| 4 | {SSDL, ORRN, VTWD} | 0.99209 | 89.224 | 0.899 |
| 5 | {SSDL, OEBA, ORRN, VTWD, VDTR} | 0.99733 | 82.305 | 0.825 |
| 6 | {SMTC, SSDL, OEBA, ORRN, VTWD, VDTR} | 0.99761 | 82.048 | 0.822 |



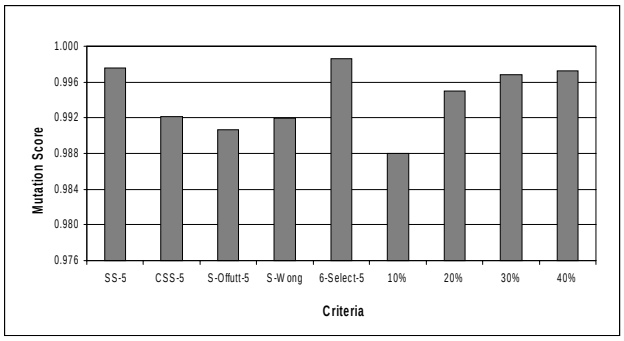

Graphic 5. Program Suite II: Sufficient Set Evolution

The results in terms of mutation score, cost reduction and cost/benefit for Selective and Randomly Mutation are summarized in Table 13 and in Graphic 6. We can observe, on average, that:

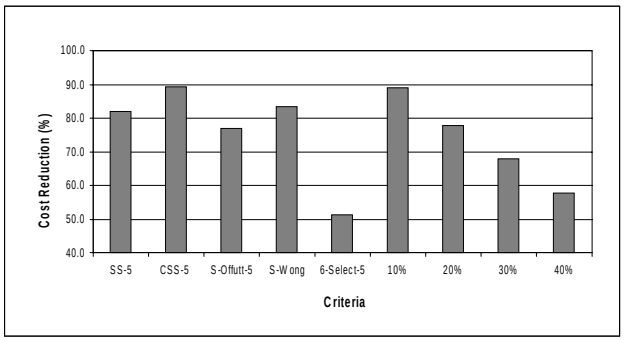
- The 6-*Selective*-5 criterion determines the greatest mutation score followed by *SS*-5.
- All criteria but 10%-*Randomly* determine a mutation score greater than 0.990.
- *CSS*-5 provides the greatest cost reduction followed by 10%-*Randomly*, *S-Wong* and *SS*-5.
- In terms of cost/benefit the best criterion is 10%-*Randomly* followed by *CSS*-5, *S-Wong* and *SS*-5.

Table 13. Program Suite II: Comparison among Selective and Randomly Mutation

| Criterion | MS | CR (%) | CR/MS |
|---|---|---|---|
| *SS*-5 | 0.99761 | 82.048 | 0.822 |
| *CSS*-5 | 0.99209 | 89.224 | 0.899 |
| *S-Offutt*-5 | 0.99066 | 77.014 | 0.777 |
| *S-Wong* | 0.99195 | 83.435 | 0.841 |
| 6-*Selective*-5 | 0.99858 | 51.340 | 0.514 |
| 10%-*Randomly* | 0.98799 | 89.045 | 0.901 |
| 20%-*Randomly* | 0.99501 | 77.700 | 0.781 |
| 30%-*Randomly* | 0.99685 | 67.835 | 0.680 |
| 40%-*Randomly* | 0.99720 | 57.675 | 0.579 |




(a)




(b)

Graphic 6. Program Suite II: Selective and Randomly Mutation: (a) Mutation Score and (b) Cost Reduction

If we favor the effectiveness of the criteria in terms of the mutation score of the adequate test sets w.r.t. a specific selective criterion against the full set of operators, we would recommend *SS-5*, *CSS-5* and *S-Wong*, respectively, as they determine mutation scores greater than 0.990 and are in the same cost/benefit range.

From now on we focus our analysis on the selective criteria. Table 14 provides a strength analysis among the selective criteria. We can observe that *SS*-5 presents the greatest strength to the other selective criteria. If we consider *ms** = 0.99 we can say that *SS*-5 $\overset{0.99}{\Rightarrow}$ *S-Offutt*-5 and that *SS*-5 is empirically equivalent to *S-Wong* and 6-*Selective*-5.

Table 14. Program Suite II: Strength Analysis

| Criteria | Mutation Score |
|---|---|
| *SS*-5 × *CSS*-5 | 1.00000 |
| *CSS*-5 × *SS*-5 | 0.98443 |
| *SS*-5 × *S-Offutt*-5 | 0.99813 |
| *S-Offutt*-5 × *SS*-5 | 0.98421 |
| *SS*-5 × *S-Wong* | 0.99993 |
| *S-Wong* × *SS*-5 | 0.99175 |
| *SS*-5 × 6-*Selective*-5 | 0.99619 |
| 6-*Selective*-5 × *SS*-5 | 0.99263 |

The uniformity of the mutation scores determined by the selective criteria for all the programs and the corresponding cost reduction for each program are illustrated in Table 15. We notice that:

- None criterion determines a mutation score equal 1.000 for any of the 5-Unix programs.
- Only *SS*-5 and 6-*Selective*-5 criteria determine mutation scores greater than 0.990 for all programs.
- The greatest cost reduction obtained with the *SS*-5, *CSS*-5, *S-Offutt*-5, *S-Wong* and 6-*Selective*-5 sets were 85.9%, 92.3%, 82.1%, 87.6% and 60.4% respectively.
- The least cost reduction obtained with the *SS*-5, *CSS*-5, *S-Offutt*-5, *S-Wong* and 6-*Selective*-5 sets were 77.9%, 84.5%, 69.5%, 79.5% and 38.5%, respectively.

Table 15. Program Suite II: Distribution of (a) Mutation Score and (b) Cost Reduction

(a)

| Program | Mutation Score | | | | |
|---|---|---|---|---|---|
| | ***SS*-5** | ***CSS*-5** | ***S-Offutt*-5** | ***S-Wong*** | **6-*Selective*-5** |
| *cal* | 0.99955 | 0.99674 | 0.99970 | 0.99961 | 0.99964 |
| *checkeq* | 0.99719 | 0.99270 | 0.98600 | 0.99713 | 0.99612 |
| *comm* | 0.99478 | 0.98725 | 0.98281 | 0.98802 | 0.99923 |
| *look* | 0.99677 | 0.98711 | 0.98838 | 0.98135 | 0.99990 |
| *uniq* | 0.99975 | 0.99667 | 0.99642 | 0.99366 | 0.99799 |
| **Average** | **0.99761** | **0.99209** | **0.99066** | **0.99195** | **0.99858** |

(b)

| Program | Cost Reduction (%) | | | | |
|---|---|---|---|---|---|
| | ***SS*-5** | ***CSS*-5** | ***S-Offutt*-5** | ***S-Wong*** | **6-*Selective*-5** |
| *cal* | 85.919 | 92.267 | 82.110 | 87.581 | 56.948 |
| *checkeq* | 81.413 | 89.190 | 74.734 | 81.478 | 60.374 |
| *comm* | 77.951 | 84.491 | 78.935 | 79.456 | 38.542 |
| *look* | 80.691 | 89.494 | 69.455 | 83.268 | 45.331 |
| *uniq* | 78.999 | 85.855 | 75.293 | 80.544 | 40.334 |
| **Average** | **82.048** | **89.224** | **77.014** | **83.435** | **51.340** |

As in Experiment I, considering mutation score, cost reduction, strength and mutation score distribution, the *SS*-5, *CSS*-5 and *S-Wong* sets would constitute the best choices. Among these, *SS*-5 provides the best mutation score, gives the best mutation score uniformity and presents the greatest strength against the other criteria. All of them provide a cost reduction over 80%.

## 6 EXPERIMENT I X EXPERIMENT II

In this section we crosscheck the selective criteria obtained for Experiment I against Experiment II and vice-versa. This give us an idea of the goodness of each selective criterion determined based on a suite of programs (or application domain) for another programs (or domains). Table 16 gives the mutant operators that compose each selective criterion. The mutation score and cost reduction obtained for the two experiments applying these criteria are summarized in Table 17.

Table 16. Selective Criteria Mutant Operators

| Criterion | Mutation Class | | | |
|---|---|---|---|---|
| | **Statement** | **Operator** | **Variable** | **Constant** |
| *SS*-27 | SWDD<br>SMTC<br>SSDL | OLBN<br>OASN<br>ORRN | VTWD<br>VDTR | Cccr<br>Ccsr |
| *SS*-5 | SMTC<br>SSDL | OEBA<br>ORRN | VTWD<br>VDTR | – |
| *CSS*-27 | SSDL | ORRN | VTWD | Ccsr |
| *CSS*-5 | SSDL | ORRN | VTWD | – |
| *S-Offutt*-27 | – | – | – | Cccr<br>Ccsr<br>CRCR |
| *S-Offutt*-5 | – | – | Vprr<br>Varr<br>VTWD<br>VDTR<br>Vsrr | – |
| *S-Wong* | STRP | OLLN<br>OLNG<br>ORRN | VTWD<br>VDTR | – |
| 6-*Selective*-27 | *OP* - {Vsrr, CRCR, Ccsr, VDTR, Cccr, SRSR} | | | |
| 6-*Selective*-5 | *OP* - {Cccr, Vsrr, Ccsr, CRCR, VDTR, ORRN} | | | |

Table 17. Selective Criteria: An Overview of the Mutation Score and Cost Reduction

| Criterion | 27-Program Suite | | 5-Program Suite | |
|---|---|---|---|---|
| | **MS** | **CR (%)** | **MS** | **CR (%)** |
| *SS*-27 | 0.99660 | 65.015 | 0.99769 | 58.711 |
| *SS*-5 | 0.98870 | 77.440 | 0.99761 | 82.048 |
| *CSS*-27 | 0.98505 | 80.031 | 0.99410 | 76.165 |
| *CSS*-5 | 0.97567 | 87.769 | 0.99209 | 89.224 |
| *S-Offutt*-27 | 0.97143 | 78.115 | 0.98829 | 70.235 |
| *S-Offutt*-5 | 0.93850 | 73.796 | 0.99066 | 77.014 |
| *S-Wong* | 0.97979 | 79.738 | 0.99175 | 83.435 |
| 6-*Selective*-27 | 0.99242 | 47.945 | 0.99864 | 53.802 |
| 6-*Selective*-5 | 0.99122 | 46.143 | 0.99858 | 51.340 |

From Table 17 we can conclude that the only sets that determine mutation scores above 0.990 for both experiments are *SS*-27, 6-*Selective*-27 and 6-*Selective*-5. Observe that *SS*-27 contains three operators (Ccsr, VDTR, Cccr) that are among the six most prevalent ones for Experiment I and four most prevalent operators (Cccr, Ccsr, VDTR, ORRN) for Experiment II. Yet, it provides a greater cost reduction than the 6-*Selective* sets for both experiments. The criterion *SS*-5 contains two operators (VDTR, ORRN) that are among the six most prevalent ones for Experiment II and one most prevalent operator (VDTR) for Experiment I. The criterion *S-Offutt*-5 is the only criterion that determines a mutation score below 0.950 when applied to Experiment I. The criterion *S-Offutt*-27 provides mutation scores greater than 0.970 for both experiments.

In Experiment I and Experiment II, considering mutation score, cost reduction, strength and mutation score uniformity we concluded that the sets *SS*, *CSS* (obtained by applying the Sufficient Procedure) and *S-Wong* yield the best results. These sets have a common set of operators as can be inferred from Table 16 and illustrated in Figure 1.

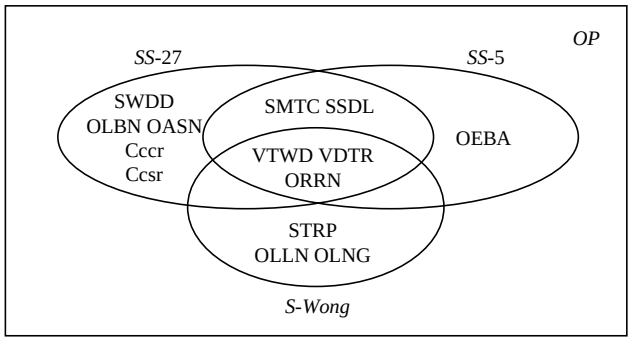



Figure 1. Commonality among *SS*-27, *SS*-5 and *S-Wong* Criteria

Observe that, except for the presence of OEBA, the sufficient set for the 5-Unix programs is a subset of the sufficient set for the suite of 27 programs. Also, observe that the operators SMTC, SSDL, ORRN, VTWD and VDTR are common to both sufficient sets and ORRN, VTWD and VDTR are common to the *S-Wong* set too.

From Table 18, we can conclude that *SS*-27 $\overset{0.99}{\Rightarrow}$ *SS*-5. From Table 9 and Table 14 it can also be concluded that *SS*-27 $\Rightarrow$ *S-Wong* and that *SS*-5 and *S-Wong* are empirically equivalent. In particular, *SS*-27 $\overset{0.99}{\Rightarrow}$ STRP, *SS*-27 $\overset{0.99}{\Rightarrow}$ OLLN and *SS*-27 $\overset{0.99}{\Rightarrow}$ OLNG and so does *SS*-5. It is important to observe that *SS*-27 $\overset{0.99}{\Rightarrow}$ OEBA, but for program *look*. By the other hand, two of the common operators (VDTR, ORRN) are among the most prevalent ones.

Table 18. Strength Analysis of the Sufficient Sets

| Experiment | Criteria | Mutation Score |
|---|---|---|
| 27-Program Suite | *SS*-27 × *SS*-5 | 0.99933 |
| | *SS*-5 × *SS*-27 | 0.98842 |
| 5-Program Suite | *SS*-27 × *SS*-5 | 0.99821 |
| | *SS*-5 × *SS*-27 | 0.99862 |

Another point that should be highlighted is related to the number of equivalent mutants generated per operator. Offutt *et al.* [15] define the *semantic size* of a fault to be the relative size of the input domain for which the program is incorrect. They suggest that the underlying goal of selective mutation is to try to only use operators that tend to produce mutants that have semantically small faults. If this model holds, their expectation would be that the selective mutants should contain a high percentage of equivalent mutants, what would impose costs to determine the equivalent mutants. In other hand, we may focus heuristic to deal with equivalent mutants just related to the sufficient operators.

In the Offutt *et al.*'s experiment it turned out the five operators in the selective set account for 57% of the equivalent mutants. In our experiments we obtained similar results. The *SS*-27 set accounts for 43.4% (1361/3136) and 56.9% (472/829) of the equivalent mutants in experiments I and II, respectively. *SS*-5 accounts for 39.6% (1242/3136) and 53.8% (446/829) of equivalent mutants in experiments I and II, respectively. *S-Offutt*-27 accounts for 15.3% and 7.2%, while *S-Offut*-5 accounts for 36.8% and 41.0% for Experiments I and II, respectively. *S-Wong* accounts for 29.6% and 45.5% for Experiments I and II, respectively. 6-*Selective*-27 accounts for 51.0% and 53.4%, while 6-*Selective*-5 accounts for 48.8% and 47.9% for Experiments I and II, respectively.

One interesting fact in this scenario is that the operators that generate the greatest percentage of the equivalent mutants, in relation to either the total number of equivalent mutants or the total number of mutants, are among the 15$^{th}$ ones. The six most prevalent ones account for 49.0% in Experiment I and 52.1% in Experiment II. We would expect then the selective operators to be among the most prevalent ones, at least for C. This would conflict with applying *N*-Selective criteria. It should also be observed that all the selective criteria account for the same range of equivalent mutants. This is a point to be further investigated.

One final point to be analyzed is the evolution of the sufficient mutant operators sets obtained in each step of the Sufficient Procedure application. Analyzing Table 7 and Table 12 and Graphic 2 and Graphic 5 we can conclude that Step 1 and Step 2 favor the mutation score while Step 3 favors the cost reduction, eliminating those mutants empirically included by others. In Step 4 we have the best cost/benefit. Step 5 and Step 6 favor the mutation score again, looking for relevant operators of each mutation class and for the high strength operators. This information should be used in a further refinement of the Sufficient Procedure proposed in this paper.

## 7 CONCLUSIONS AND FURTHER WORK

We report in this paper two experiments toward the determination of sufficient mutant operators for C, in the same line of Offutt *et al.*'s study for FORTRAN [15]. In the scope of these case studies a procedure for the determination of a sufficient mutant operators set for C language, in the context of *Proteum* testing tool, was proposed. The Sufficient Procedure [3][4], as it was named, aims at providing a systematic way to determine a selective criterion based on Mutation Testing. The proposed procedure synthesizes the guidelines, discussed in this paper, for determination of sufficient mutant operators we have devised motivated by previous results in the area. The guidelines explore concepts such as mutation score determined by a specific operator, inclusion relation among the operators, strength and mutation type. Along this study

a comparison with the most relevant previous works on this subject was also carried out.

The Sufficient Procedure application leaded to a considerable reduction on the number of available operators (71) in *Proteum*. The sufficient mutant operators sets obtained provided a high adequacy degree w.r.t. Mutation Testing: the mutation scores were above 0.995. Considering the application cost, in terms of number of mutants, the reductions were, on average, above 65%

In both experiments, considering the mutation score, cost reduction, strength and mutation score distribution, the sufficient operator sets determined by the application of the Sufficient Procedure would be among the best choice. They presented the greatest mutation scores, empirically included the other selective criteria, presented an excellent mutation score uniformity among the programs and determined the greatest strengths against the other selective criteria.

The computational cost to determine a sufficient mutant operators set may strongly influence the choice of a specific approach. The aim is to define a pragmatically, low-cost, domain-independent approach for the determination of such set. For instance, Wong *et al*.'s selective set is domain and program independent.

One important point to be looked at, related to the guidelines i and ii, are the good results obtained with the constrained sufficient sets (*CSS*), which include the more representative operator of each mutation class. These sets constitute a very good start point to build up a sufficient mutant operators set.

Another point is that the proposed Sufficient Procedure's structure makes possible that the determination of the sufficient set be done and applied in an incremental way, according to system criticality and time and budget constraints. If we apply just the guidelines i to iv, i.e., just the operators obtained in Step 4, we would have a high mutation score, with a low application cost. The other two steps related to guidelines v and vi, favor the test effectiveness, in terms of mutation score, but compromise the cost/benefit relation.

In this work we did not take in account other costs associated to Mutation Testing as well as the effectiveness. For instance, Mresa *et al*. [13] have also investigated sufficient FORTRAN mutant operators using Mothra, taking in account the mutation scores provided by the individual operators and their associated costs (including both test set generation and equivalent mutant detection) in order to determine the most efficient operators. According to the authors, the results show that the use of the efficient operators can provide significant gains for Selective Mutation if the acceptable mutation score is not very close to 1.000, otherwise, Randomly Selected Mutation provides a more efficient strategy than a sufficient set of operators. Mresa *et al*. have also raised the point it can not be assumed that a test set that kills 99% of the mutants killed by an adequate test set is able to detect 99% of the real faults that are detected by the adequate test set. Our measure of effectiveness considered just the mutation score of the selective criteria.

Giving the considerations above we are motivated to further investigate and refine the sufficient sets, taking in consideration operator cost, in terms of number of generated mutants, of equivalent mutants and of number of test cases. Also, further investigation analyzing the abilities of the selective criteria to detect real faults will be carried out.

As mentioned before, the sufficient set may be dependent on the application domain and on the programs used, i.e., the specific characteristics of each program (or suite of programs). This is also a point to be further investigated. Applying the procedure in other programs at different domains will generate a knowledge base that may be used for improving the sufficient set.

Further studies are been planned to investigate the scalability of these results to larger programs. We are also interested in conduct a broad selection of programs, from different application domains, to replicate this study, in order to make the results presented so far, more significant.

A similar study has been carried out at the integration testing level, considering the Interface Mutation criterion [8] and using *Proteum/IM* [7] – a tool that supports the testing of C programs at the integration level, and similar results, to appear in a forthcoming paper, were obtained.

## ACKNOWLEDGEMENTS

The authors would like to thank the Brazilian funding agencies CAPES, FAPESP and CNPq for their support to the research activities carried out by the Software Engineering Group at ICMC/USP, São Carlos, Brazil.

## REFERENCES

[1] A.T. Acree, T.A. Budd, R.A. DeMillo, R.J. Lipton, F.G. Sayward. *Mutation Analysis*. Technical Report GIT-ICS-79/08, Georgia Institute of Technology, Atlanta, GA, September 1979.

[2] H. Agrawal, R.A. DeMillo, R. Hathaway, W. Hsu, Wy Hsu, E.W. Krauser, R.J. Martin, A.P. Mathur, E.H. Spafford. *Design of Mutant Operators for the C Programming Language*. Technical Report SERC-TR-41-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, March 1989.

[3] E.F. Barbosa. *A Contribution for the Determination of a Sufficient Mutant Operators Set for C-Program Testing*. MSc. Thesis, ICMC/USP, São Carlos, SP, Brazil, November 1998 (in Portuguese).

[4] E.F. Barbosa, A.M.R. Vincenzi, J.C. Maldonado. A Contribution for the Determination of a Sufficient Mutant Operators Set for C-Program Testing. In *Proceedings of the 12th Brazilian*

*Symposium of Software Engineering*, pages 103-120, Maringá, PR, Brazil, October 1998 (in Portuguese).

[5] T.A. Budd, R.A. DeMillo, R.J. Lipton, F.G. Sayward. Theoretical and Empirical Studies on Using Program Mutation to Test the Functional Correctness of Programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 220-233, New York, NY, January 1980.

[6] M.E. Delamaro, J.C. Maldonado. Proteum: A Tool for the Assessment of Test Adequacy for C Programs. In *Proceedings of the Conference on Performability in Computing Systems (PCS 96)*, Brunswick, NJ, July 1996.

[7] M.E. Delamaro, J.C. Maldonado. Interface Mutation: Assessing Testing Quality at Interprocedural Level. In *Proceedings of the 19th International Conference of the Chilean Computer Science Society (SCCC)*, pages 78-86, November 1999.

[8] M.E. Delamaro, J.C. Maldonado, A.P. Mathur. Interface Mutation: An Approach to Integration Testing. *IEEE Transactions on Software Engineering* (to appear).

[9] R.A. DeMillo, R.J. Lipton, F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer*, 11(4), pages 34-41, April 1978.

[10] R.A. DeMillo, D.S. Gwind, K.N. King, W.N. McKraken, A.J. Offutt. An Extended Overview of the Mothra Testing Environment. In *Proceedings of the 2nd Workshop on Software Testing, Verification and Analysis*, Banff, Canada, July 1988.

[11] A.P. Mathur. Performance, Effectiveness, and Reliability Issues in Software Testing. In *Proceedings of the 15th Annual International Computer Software and Applications Conference*, pages 604-605, Tokio, Japan, September 1991.

[12] A.P. Mathur, W.E. Wong. An Empirical Comparison of Data Flow and Mutation-Based Test Adequacy Criteria. *The Journal of Software Testing, Verification and Reliability*, 4(1), pages 9-31, March 1994.

[13] E.S. Mresa, L. Bottaci. Efficiency of Mutation Operators and Selective Mutation Strategies: an Empirical Study. *The Journal of Software Testing, Verification and Reliability*, 9(4), pages 205-232, December 1999.

[14] A.J. Offutt, G. Rothermel, C. Zapf. An Experimental Evaluation of Selective Mutation. In *Proceedings of the 15th International Conference on Software Engineering*, pages 100-107, Baltimore, MD, May 1993.

[15] A.J. Offutt, A. Lee, G. Rothermel, R.H. Untch, C. Zapf. An Experimental Determination of Sufficient Mutant Operators. *ACM Transactions on Software Engineering Methodology*, 5(2), pages 99-118, April 1996.

[16] S. Rapps, E.J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4), pages 367-375, April 1985.

[17] A.M.R. Vincenzi. *Subsidies to the Establishment of Mutation Based Testing Strategies.* MSc. Thesis, ICMC/USP, São Carlos, SP, Brazil, November 1998 (in Portuguese).

[18] E.J. Weyuker. The Cost of Data Flow Testing: An Empirical Study. *IEEE Transactions on Software Engineering*, 16(2), pages 121-128, February 1990.

[19] E.J. Weyuker, S.N. Weiss, R.G. Hamlet. Comparison of Program Testing Strategies. In *Proceedings of the 4th Symposium on Software Testing, Analysis and Verification*, pages 154-164, Victoria, British Columbia, Canada,October 1991.

[20] W.E. Wong, J.C. Maldonado, A.P. Mathur. Mutation versus All-Uses: An Empirical Evaluation of Cost, Strength, and Effectiveness. *Software Quality and Productivity – Theory, Practice, Education and Training*, Hong Kong, December 1994.

[21] W.E. Wong, A.P. Mathur. Reducing the Cost of Mutation Testing: An Empirical Study. *The Journal of Systems and Software*, 31(3), pages 185-196, December 1995.

[22] W.E. Wong, J.C. Maldonado, M.E. Delamaro, S.R.S. Souza. A Comparison of Selective Mutation in C and FORTRAN. In *Proceedings of the Workshop of the Validation and Testing of Operational Systems Project*, pp. 71-84, Águas de Lindóia, SP, Brazil, January 1997.

[23] W.E. Wong, J.C. Maldonado, M.E. Delamaro. Reducing the Cost of Regression Testing by Using Selective Mutation. In *Proceedings of the 8th International Conference on Software Technology (CITS)*, pages 11-13, Curitiba, PR, Brazil, June 1997.